UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

MASTER IN COMPUTER SCIENCE AND TELECOMMUNCATION ENGINEERING

Desarrollo de un sistema de toma de datos genérico con aplicación al campo de la física nuclear

Autor: Cayetano Santos Buendia
Tutor: Eduardo Boemo Scalvinoni

30 de agosto de 2013

Resumen

Esta memoria resume el trabajo efectuado durante los últimos meses con el objetivo de desarrollar un demostrador funcional completo de un sistema de toma de datos genérico. Dicho sistema tiene como objetivo principal el de instrumentar un experimento real, encontrando así una primera aplicación directa en el campo de la física nuclear experimental. El sistema integra elementos separados existentes, a partir de los cuales se ha desarrollado todo el código embebido y software necesario para su funcionamiento.

Indice

Índice de figuras

Parte I
Presentación

1. Introducción

Dentro del ámbito de la investigación científica experimental, existe una necesidad recurrente relacionada con la instrumentación de señales provenientes de sensores arbitrarios. Dicha instrumentación implica la posibilidad de poder adquirir y procesar digitalmente dichas señales, extrayendo las informaciones mas relevantes. Este tipo de problemática es frecuente en campos muy dispares, pero posee características similares en todos ellos, como son la necesidad de poder reprogramar el sistema, de configurarlo adecuándolo a nuevas necesidades. Así, se requieren habitualmente sistemas genéricos de toma de datos, polivalentes y flexibles que puedan ser adaptados a una aplicación en concreto de manera sencilla. Esto implica el hecho frecuente de poder compartir el sistema dentro de un proyecto de colaboración, de forma que se pueda adaptar la funcionalidad de base implementada originalmente a otros campos de instrumentación donde se deba procesar señales de características parecidas.

De esta manera, y con el objetivo principal de responder a aplicaciones relacionadas con la instrumentación de señales rápidas en el campo de la física experimental, caracterizadas por un ancho de banda de varias decenas de megahercios (MHz), estos sistemas habrán del cumplir un cierto número del requerimientos. Así, se querrá que puedan gestionar hasta una decena de canales electrónicos [3], con un ancho de banda de varias decenas de MHz y una precisión y dinámicas de medida elevadas (unos 14 bits) [1]. Además, se requerirá habitualmente mantener un coste por canal reducido como requerimiento fundamental, así como poder integrar capacidades de procesamiento en línea configurables [2] y modificables por un usuario del sistema sin que éste deba adquirir conocimientos específicos, lo que supondría una penalización para el propio instrumento. Por último, en ocasiones se exigirá como requerimiento adicional el poder disponer de un sistema portátil, lo que facilitará su uso en entornos difíciles [4].

Dentro de este contexto, y teniendo como objetivo fundamental el resolver un problema real, se ha desarrollado un demostrador completo de sistema de adquisición y procesado de datos. En efecto, este proyecto tiene una aplicación práctica inmediata como sistema de medida y procesamiento de señales provenientes de detectores en física nuclear en experimentos reales [5]. Así, uno de los objetivos de desarrollar este demostrador completo de sistema de toma de datos tiene como finalidad principal la de poder equipar un experimento existente. Las tomas de datos se efectuarán durante campañas de medida que tendrán lugar durante la segunda mitad del 2013. Esta memoria tiene por objeto el describir las etapas seguidas en el desarrollo de este sistema, que ha pretendido ser lo más genérico posible con el objetivo de poder ser extendido y ampliado de manera sencilla a otro tipo de aplicaciones.

2. Componentes y etapas del desarrollo

Este proyecto se ha podido llevar a cabo teniendo en cuenta una metodología diferente de la seguida habitualmente en este tipo de proyectos, basados en la concepción íntegra (esquemático, pcb, etc.) de la electrónica necesaria. De manera alternativa, en esta ocasión se ha desarrollado un planteamiento a partir de la existencia de un hardware comercial económico, genérico y configurable. Este hardware se presenta bajo la forma de varios módulos electrónicos que pueden ser combinados entre ellos con el objetivo de implementar la funcionalidad requerida.

Siguiendo este principio, el sistema de toma de datos se ha fundamentado en tres unidades o módulos comerciales (COTS [1]) claramente diferenciados [7]

- en primer lugar, un módulo procesador CPU, que actuará como colector datos, basado en un SoC ARM, capaz de ejecutar un sistema operativo Linux estándar que requiera pocos recursos y que permitirá conectar periféricos ordinarios (usb, hdmi, eSATA). Este dispositivo permite además el desarrollo de aplicaciones programables en lenguajes de alto nivel y posee la potencia de cálculo necesaria para gestionar todas las tareas de gestión de configuración y enlace por red (sección 3.3)
- en segundo lugar, se ha empleado un módulo basado en una FPGA de gama media Spartan 6 y que integra un puerto usb esclavo, permitiendo su conexión al módulo CPU anterior. Esta plataforma integrara un procesador software, configurable en lenguaje C, junto con aceleradores de bajo nivel concebidos en lenguajes VHDL / c-HLS, lo que confiere al sistemas capacidades de reconfiguración mixtas tanto hardware como software (sección 3.2)
- por último, el módulo anterior vendrá acoplado a un tercer módulo de alta tasa de muestreo de señales (ADC) de hasta 16 canales que hará las veces de frontal analógico (sección 4)

Esta memoria describe en detalle cuales han sido las etapas seguidas y como han sido combinados dichos módulos de forma y manera adecuada con el objetivo de implementar un sistema completo y genérico de toma de datos, al mismo tiempo que se ha desarrollado todo el código necesario para su funcionamiento. Este desarrollo incluye entre otros los siguientes elementos

- el código software de nivel aplicativo de control y gestión del sistema
- el código embebido de configuración, adquisición y procesado en tiempo real de las señales muestreadas
- la integración en el sistema los aceleradores hardware necesarios específicos a la aplicación requerida

En resumen, con este trabajo se pretende presentar una implementación genérica de este demostrador, que pretende ser abierto y polivalente, presentando para terminar un ejemplo de aplicación práctico real. Esta primera aplicación podrá posteriormente ser adaptada a situaciones arbitrarias basándose en la naturaleza configurable y modular del conjunto.

[1] `http://goo.gl/2xRacm`

3. Modulos utilizados

El tipo de aplicaciones que se pretendan cubrir con este sistema determina en gran parte la selección de los módulos electrónicos que se han elegido. En primer lugar, las características de las señales que se desea procesar llevan a el tipo de muestreo que se debe realizar (frecuencia, ancho de banda, resolución dinámica, número de canales, etc.), lo que a su vez fija las características del módulo de muestreo. Así, el decidirse por un módulo mezzanine en formato FMC [8] (FMC112) limita las posibilidades de elección de una placa con recursos programables (Shuttle LX1). A su vez, las características de esta placa (bus, librerías, etc.) llevan a la elección del hardware colector de datos. Este colector (Cubox), debe ser compatible con lo anterior, siendo a su vez el conjunto coherente. Las características de los módulos elegidos, así como su funcionalidad dentro de este sistema se detallan a continuación.

3.1. FMC112

Siguiendo el razonamiento anterior, y a partir del conocimiento de las señales de entrada al sistema, se ha decidido elegir éste módulo[2]. Se desea procesar señales impulsionales de un ancho de banda de hasta 40 MHz, con un rango dinámico extenso con amplitudes variando en un factor 4000, lo que equivale a un número mínimo efectivo de bits de 12. Estas dos características fijan el tipo de adc que se deba utilizar: frecuencia de muestreo superior a 100 MHz y número de bits de 14. Además, se pretende trabajar con un sistema multicanal capaz de procesar al menos 10 señales, con acoplo DC para aprovechar el rango dinámico del adc y control de offset. Por último, se sabe que la amplitud de las señales oscilará en un rango de entre 1 y 2 voltios. El módulo FMC112 [11] posee todas estas características, por lo que su elección visto el precio final por canal el mas que razonable. Al venir dado este módulo en formato FMC, se ha buscado una plataforma portadora con este mismo estándar.

3.2. Shuttle LX1

Este segundo módulo[3] es compatible con el anterior, permitiendo acoplar ambos de manera sencilla [10]. Posee un conector FMC LPC conforme a esta norma, cuyas señales han sido ruteadas directamente a una FPGA Spartan 6 de tamaño medio. Además, también posee una memoria DDR2 de 128 MB y permite alimentar a la tarjeta FMC112 mezzanine a partir de una alimentación de 12 V externa.

Este módulo, entre todos los posibles con recursos similares, posee tres características que lo hacen idóneo[4]. En primer lugar, posee un precio muy reducido (200 Euros). En segundo lugar, ofrece un interfaz usb 2 de tasa de transferencia de hasta 25 MB/s, medidos experimentalmente. Como punto mas positivo, viene acompañado de la librería FrontPanel (FP) necesaria para hacerlo funcionar rápidamente. Estas librerías incluyen tanto la parte software como hardware, y suponen todo un entorno para la puesta en funcionamiento de la tarjeta. Sin embargo, como se verá mas adelante, estas librerías han sido empleadas tan solo parcialmente en este proyecto por razones prácticas de aprovechamiento del espacio disponible dentro de la FPGA, habiéndose añadido un nivel superior a un subconjunto de la librería.

En el contexto de este proyecto, en el cual es necesario poder hacer funcionar el sistema de toma de datos tanto desde un pc de arquitectura Intel estándar como desde una plataforma ARM, es de suma importancia el disponer de una versión de la librería estática para esta última arquitectura.

[2] `www.4dsp.com/FMC112.php`

[3] `http://www.opalkelly.com/products/xem6006/`

[4] `http://goo.gl/4vOhkh`

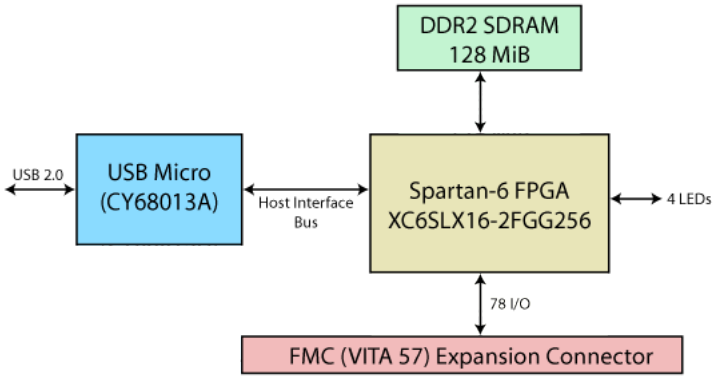


Figura 2: Diagrama de bloques del módulo Shuttle LX1 de OpalKelly

Este es el caso desde hace seis meses, en el que la sociedad OpalKelly provee una versión binaria de las librerías para plataformas ARM. Esta versión se ha venido utilizando con éxito en este proyecto.

Como punto negativo de esta placa cabe reseñar el tamaño de la FPGA Spartan 6, lo que limita y condiciona todo el desarrollo efectuado a partir de ella. La metodología seguida teniendo en cuenta esta limitación se detalla en las siguientes secciones de esta memoria.

3.2.1. Librerias FrontPanel (Hardware)

Las librerías FrontPanel[5] (FP) son autocontenidas en el sentido en el que proveen todos los elementos necesarios al desarrollo de aplicaciones sobre la plataforma Shuttle LX1, así como con otras similares de la sociedad OpalKelly [9] .

Desde el punto de vista del hardware en la FPGA, FP implementa un controlador usb bajo la forma de una netlist. Esto evita al usuario cualquier implementación de bajo nivel respecto al bus de salida de datos, haciéndola transparente. A partir de aquí, se dan cuatro tipos de 'endpoints': triggers, wires, pipes y su generalización, lo llamados block-throttled pipes (BTP). Cada uno de ellos funciona de una manera distinta y tienen un propósito concreto para cada tipo de aplicación, aunque no es el objetivo de esta memoria el detallarlos. Una visión modular esquemática de su funcionamiento se puede observar en la figura 3.

Tanto el controlador como cada uno de los endpoints se da en la forma de un módulo hdl alrededor de una netlist, que se deberá de instanciar en el diseño, siendo posible incluir tantos como se deseen. Cada uno de ellos configura con una dirección, que lo identificará de manera única. Existirán además versiones de salida y de entrada, siempre desde el punto de vista de la FPGA.

3.2.2. Librerias FrontPanel (Software)

Los recursos anteriormente citados tienen su reflejo en una interfaz software (API) con pasarelas para varios lenguajes de programación de nivel alto (c, python, java, matlab). De esta manera se provee un método sencillo de comunicación entre el nivel de aplicación y el nivel hardware (endpoints). Esta API está incluida en una biblioteca de funciones que viene dada bajo la forma de una librería en dos versiones, una dinámica que puede ser cargada en memoria durante la ejecución del programa y otra estática.

[5] `http://goo.gl/Ufr3B0`

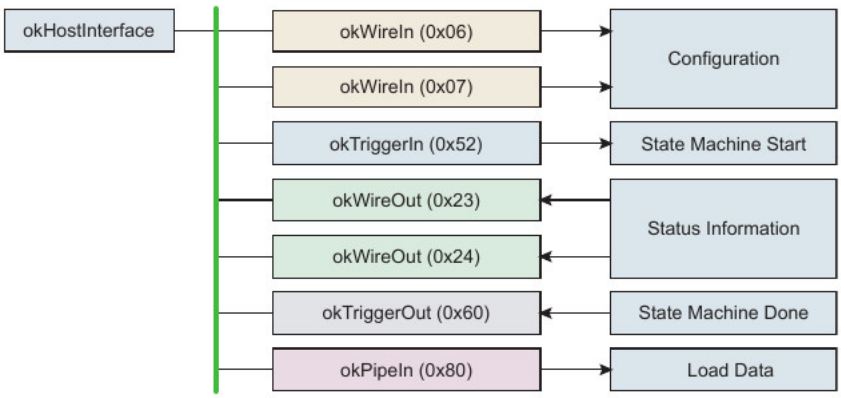


Figura 3: 'Endpoints' de las librerías FrontPanel

El funcionamiento de esta API es similar bajo cualquiera de los lenguajes anteriormente citados: la librería dinámica es cargada en memoria, pudiéndose ejecutar llamadas a las funciones que contiene. Estas funciones serán de tipo inicialización y gestión de periféricos, configuración e intercambio de datos mediante rutinas de lectura y escritura sobre los endpoints definidos en el diseño hardware. Esto se consigue mediante el indentificador asignado a cada uno de ellos anteriormente.

Por último, señalar que FP ofrece posibilidades extras de construccion de interfaces gráficas y otros recursos redundantes dentro del contexto de este desarrollo, que pretende ser lo mas abierto, simplificado y modular posible.

3.3. Cubox

Como plataforma colectora[6] de datos se ha escogido una placa basada en un procesador ARMv7. Entre las muchas posibilidades existentes, se ha escogido ésta al poseer características únicas. En primer lugar, cumple los requerimientos fundamentales para esta aplicación, como son el poseer un tamaño y un precio reducidos (100 Euros). Además, incluye los recursos materiales necesarios. Entre ellos cabe destacar

- 2 GB de memoria ram DDR3 @ 800 MHz de 32 bits, que son utilizados por el software como memoria tampón temporal previo al envío de los datos colectados por la interfaz de red o su almacenamiento en local en disco
- procesador de 800 MHz dual issue ARM PJ4, VFPv3, wmmx SIMD and 512KB L2 cache, con suficiente potencia de cálculo como para gestionar adecuadamente un flujo de datos usb 2 en continuo desde el periférico, así como para realizar un procesado en línea
- capacidades gráficas avanzadas de reproducción de vídeo de hasta 1080p, que permiten una visualización en local de los resultados de la toma de datos
- 3 watios de consumo de potencia, relevantes a la hora de considerar la disipación térmica del dispositivo

Como añadido, posee también un puerto de salida hdmi / vga para poder conectar un monitor si fuera necesario; dos puertos usb 2 (extensibles por un hub) que permiten utilizar un conjunto de

[6] `http://solid-run.com`

Figura 4: Módulo Cubox de SolidRun

teclado / ratón inalámbricos, añadir una interfaz de red inalámbrica y conectar el periférico que se desee controlar. Además, posee una interfaz de red cableada de 1 Gbit ethernet, un emplazamiento para memoria micro SD de hasta 32 GB que hará la vez de disco duro y un puerto micro usb para conexiones remotas por consola, que permitirán actualizar la memoria flash de la plataforma e instalar un sistema operativo. Por último, tiene como característica especial el hecho de poseer un puerto eSata de 3 Gbps, lo que nos ofrece la posibilidad de almacenar datos de la adquisición localmente sobre discos de varios TB de capacidad.

Por último, este hardware posee un soporte software amplio y validado a través de las muchas aplicaciones que se han desarrollado durante los últimos años. Estas aplicaciones suponen una garantía en cuanto a la calidad del hardware, además de haber motivado el desarrollo de drivers optimizados a medida que soporten todo lo anterior. El soporte técnico ofrecido también es considerable, y la comunidad de desarrolladores es activa.

Cubox (figura 4) se utiliza en esta aplicación como plataforma en la que desarrollar el software necesario para controlar y adquirir datos desde un periférico usb. Para ello se utiliza un sistema operativo Arch Linux, ligero, rápido y eficiente, que se ejecuta desde una memoria microSD. El soporte para el hardware es muy bueno, y permite desarrollar código en lenguaje Python, compatible con otras plataformas en caso de necesitarlo. Los datos del periférico son bien transferidos sobre la interfaz de red (inalámbrica o no), bien almacenados en local sobre un disco duro externo por el puerto eSata en caso de que la memoria micro-SD de 32 GB no fuera suficiente. En caso de no poder acceder a la toma de datos durante periodos prolongados de tiempo, esta arquitectura permite poder controlar la toma de datos de manera remota por red y almacenar datos durante un tiempo suficiente.

4. Metodología de trabajo

En paralelo al desarrollo realizado, y como complemento de éste, la ejecución de este proyecto ha producido un efecto beneficioso anexo en cuanto a la metodología de trabajo que se ha seguido de cara a su ejecución en los plazos requeridos. Así, este proyecto ha pretendido ser planteado desde una óptica novedosa, lo que ha conducido a una metodología de trabajo también original. Tras proceder a un análisis inicial de los requerimientos necesarios para gestionar el proyecto, ha podido ponerse en evidencia que en el curso de este desarrollo debería de hacerse frente a una variedad importante de metodologías, lenguajes y herramientas, lo que conlleva una carga importante en todo lo relativo a su organización.

Así, por una parte se ha procedido a realizar una primera etapa de documentación y prospección bibliográfica, que ha llevado a dar forma a este proyecto. Posteriormente, se ha debido de proceder a concebir código (tanto embebido como software) en varios lenguajes de programación distintos: vhdl, c, python. Por otra parte, ha sido necesario redactar una memoria de proyecto con sus fuentes en LaTeX, automatizando su exportación hacia pdf, html, etc. Ha sido necesario además documentar todo lo anterior, organizando y gestionando las diversas etapas seguidas. Durante el curso del proyecto, se ha pretendido además gestionar y organizar las anotaciones, comentarios y ficheros adjuntos a cada etapa, integrándolos bien a la documentación, bien a la propia memoria del proyecto.

Con la finalidad de organizar todo lo anterior, se ha optado por una metodología de trabajo lo mas ergonómica y sencilla posible, realizando un ejercicio interesante de gestión global de proyecto de desarrollo en paralelo al desarrollo principal objeto de esta memoria. La elección de dicha metodología resulta relevante desde el momento en el que las tareas a realizar aumentan y es necesario mantener una trazabilidad que redunde en beneficio del proyecto.

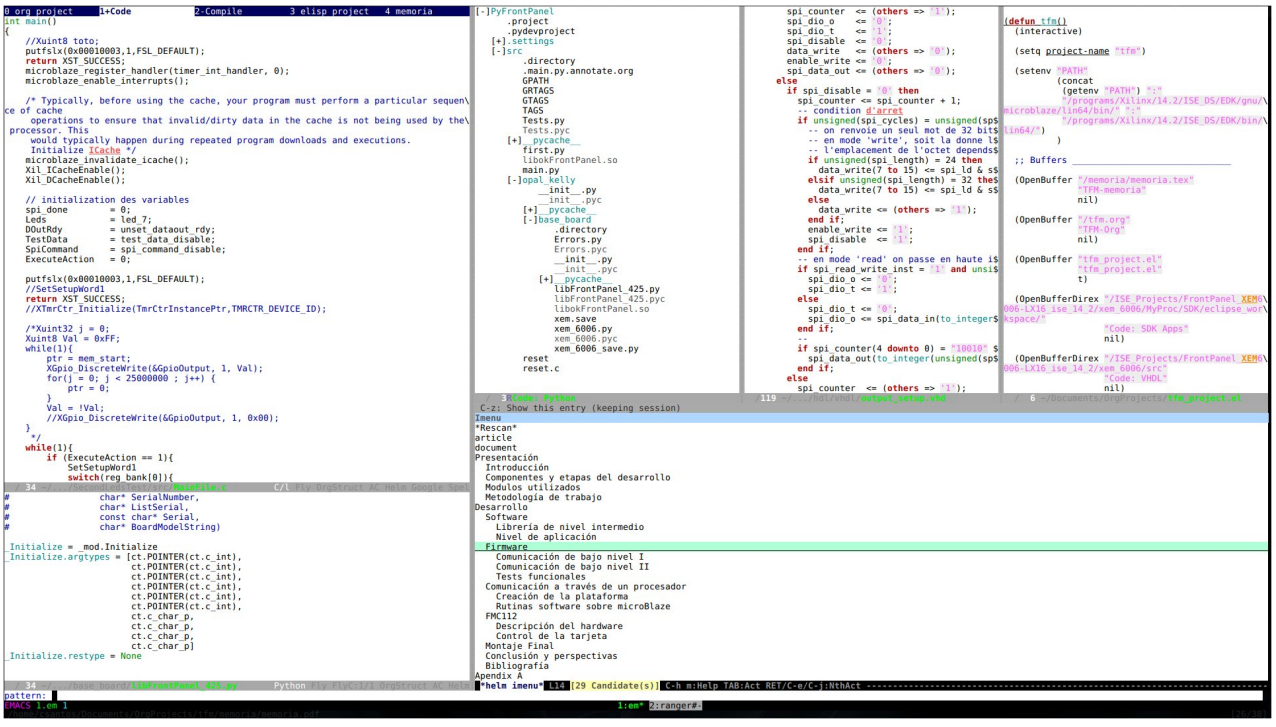


Figura 5: Gestión multiproyecto con GNU-Emacs. Se puede observar como desde un mismo entorno de trabajo pueden ser configuradas y gestionadas varias tareas de programación, organizacion. etc.

De esta manera, se ha procedido a seleccionar como entorno de trabajo la herramienta GNU Emacs, al ser ésta una herramienta que resume e integra todos los requerimientos anteriores. De esta manera ha podido seguirse una metodología de trabajo consistente en, por una parte, la gestión, documentación y creación de una agenda del proyecto (modo Org[7]), incluyendo la manipulación de anotaciones, referencias, enlaces y ficheros adjuntos adecuados en cada etapa. Por otra parte se ha facilitado el desarrollo del código necesario a su desarrollo. Este último punto se simplifica en gran medida al poder trabajar dentro de un único entorno de desarrollo con soporte avanzado para los lenguajes necesarios, incluyendo capacidades características de un entorno evolucionado (emacs code browser), la gestión de navegación de fuentes, semánticas (cedet), corrección, compilación, ejecución, debug, etc.. Por último, el hecho de poder configurar el entorno gráfico visual (capacidades multi ventana) a las necesidades del proyecto, adaptándolo a un entorno de desarrollo multi lenguaje (entorno tabulado) y automatizando tareas recursivas (lenguaje elisp) ha aportado un grado de ergonomía difícilmente alcanzable de otra manera. La segunda parte de esta memoria permite ver de manera más en detalle como funciona esta estructura de trabajo, que ha resultado, en paralelo con el desarrollo principal, una experiencia interesante de organización y gestión multiproyecto.

[7] `http://orgmode.org`

Parte II
Desarrollo

Durante la segunda parte de esta memoria se presentara el desarrollo efectuado a partir del contexto, los requerimientos, las ideas expuestas y los módulos descritos en la primera parte. Este desarrollo se compone de dos partes principales, y se basa a un nivel fundamental en las librerías FrontPanel. En primer lugar se presenta el desarrollo software realizado, a dos niveles distintos, y de manera más o menos estándar. A continuación se detallará la concepción de código embebido que se ha llevado a cabo con el objetivo de implementar la funcionalidad requerida para la aplicación, y que supone el punto central de este proyecto.

1. Software

El desarrollo software realizado para este proyecto se ha descompuesto en dos niveles. En un nivel superior se encuentra el código de nivel de aplicación, y que debe de ser desarrollado en función de la misma. En un nivel inferior aparecen las rutinas de nivel bajo de comunicación a través del bus usb con el módulo periférico. Este segundo nivel aparece explicado en la sección siguiente de esta memoria.

1.1. Librería de nivel intermedio

La aplicación objeto de este desarrollo de sistema de toma de datos requiere un alto grado de flexibilidad. Este requerimiento interviene tanto a nivel hardware como software. Así, se desea poder ofrecer la posibilidad de desarrollar varias aplicaciones distintas por usuarios diferentes a partir de una base común a todas ellas. De esta manera se simplifica el uso de este sistema en proyectos distintos, facilitando la reutilización, la actualización del código y la modularidad del conjunto. Por ejemplo, esta concepción modular del sistema de toma de datos posibilita un desarrollo a medida para una aplicación particular, sin que la gestión global de las fuentes se complique en exceso.

Por esta razón, a la hora de concebir la arquitectura software de la aplicción se ha optado por independizar claramente la comunicación de nivel intermedio (estando el bajo nivel ya dado) de la aplicación de nivel de usuario. El nivel intermedio debería de ser estable mientras que el nivel de aplicación es susceptible de modificaciones más frecuentes por usuarios independientes. Siguiendo este principio, se ha optado por no hacer uso de la API de alto nivel contenida en las librerías FrontPanel, implementando una librería dinámica a medida de nivel intermedio a partir de la capa de nivel mas bajo contenida en la librería FrontPanel. Esta decisión de arquitectura resulta de gran importancia en el caso de deber gestionarse una multitud de sistemas distribuidos en emplazamientos remotos.

Esta manera de proceder tiene ventajas evidentes respecto a los requerimientos de este sistema. Al desarrollar nuestra propia biblioteca de funciones la aplicación final sigue pudiendo ejecutar dinámicamente rutinas de comunicación con el periférico. Al mismo tiempo, éstas rutinas tienen un significado y una interpretación simplificada, haciendo completamente transparente a la aplicación cualquier detalle de funcionamiento de nivel bajo. La aplicación tan solo ha de ejecutar llamadas genéricas fácilmente interpretables, de modo que la bilbioteca que hemos desarrollado gestiona toda la posible complejidad existente en relación con la comunicación con el periférico. Esto posibilita el desarrollo de aplicaciones de nivel superior de manera independiente del hardware por los usuarios del sistema.

Más en detalle, la API FrontPanel define una serie de funciones de interfáz de identificación de periférico (*GetDeviceSerial*, *GetDeviceVersion*, etc.), de apertura (*OpenbySerial*) y modificación de identificativos (*SetID*, *SetEeprom*, etc.). Además, permite la configuración de la FPGA contenida en el módulo Shuttle LX1 (*ConfigureFPGA*, *Reset*, etc.) a partir de un diseño contenido en un fichero de configuración (*.bit) Todas estas funciones de gestión de bajo nivel implican un nivel de detalle al que no se desea exponer a la capa de aplicación. Con este objetivo, se ha desarrollado en lenguaje c una interfaz de rutinas de nivel más elevado, como se muestra en la figura 6.

Figura 6: Librería dinámica a medida de nivel intermedio

La API FrontPanel viene dada en tres ficheros: la librería binaria *.so, y los ficheros fuente *.cpp y de encabezados y definiciones *.h. A partir de aquí se ha desarrollado una librería a medida (x6006-lib) de funciones de alto nivel. Las tres funciones principales son las de identificación e inicialización, configuración y finalización del periférico detectado en el bus usb.

- *Initialize* Esta función carga en memoria la librería FrontPanel, inicializa un puntero interno de referencia al periférico, devolviendo a la aplicación informaciones sobre el número de periféricos detectados y su configuración actual. Además, devuelve una serie de mensajes de error en caso de detectar algún problema.
- *Configure* En esta segunda etapa se envía el fichero de configuración y se verifica que el proceso se ha ejecutado correctamente, devolviendo las correspondientes informaciones en caso contrario.
- *Finalize* Esta función es responsable de ejecutar los pasos necesarios antes de finalizar la aplicación, liberando recursos.

En una segunda categoría, see han previsto igualmente funciones adicionales de gestión en el caso de querer desarrollar una aplicacón que utilice varios periféricos (*GetDeviceID*, *SetDeviceID*), así como otras de parametrización de la toma de datos (*SetTimeOut*, *EnableAsynchronousTransfers*, *SetBTPipePollingInterval*, *GetLastTransferLength*).

Por último, se han añadido rutinas de lectura y escritura de datos hacia los endpoints definidos en el diseño. La llamada de estas rutinas debe de ser compatibe con el contenido del diseño en el fichero de configuración *.bit, e identifica a cada endpoint a través de una dirección única, especificada en dicho diseño. Las dos rutinas de comunicación hacia endpoints de tipo BTP son *WriteToPipeIn* y *ReadFromPipeOut*, siendo éstas las únicas que se han utilizado en este proyecto. Las rutinas de comunicación hacia otro tipo de endpoints (wires, triggers) también han sido incluidas, aunque no sean estrictamente necesarias para este desarrollo.

Esta biblioteca puede ser recompilada sobre varias plataformas (intel, arm) de forma sencilla, lo que facilita su portabilidad. Supone un único punto de entrada hacia el hardware para cualquier aplicación que desee desarrollarse, que no habrá de ocuparse de los detalles de funcionamiento de bajo nivel. Contiene todas las llamadas necesarias de configuración y adquisición, definiéndose la funcionalida que implementen en un nivel superior. A partir de aquí puede comenzar a definirse una aplicación en un lenguaje que soporte llamadas a librerías dinámicas genéricas. Un ejemplo de aplicación se muestra en la siguiente sección de esta memoria.

1.2. Nivel de aplicación

Con el objetivo de mostrar un posible uso de la librería detallada en la sección anterior, pero también con la finalidad de poder emplear este sistema de toma de datos en una aplicación real se ha desarrollado el software de nivel de aplicación necesario para su funcionamiento. En esta memoria se presentará el software desarrollado únicamente con la finalidad de mostrar una vista de conjunto del sistema de toma de datos, pero no se entrará en detalles de su estructura o comportamiento, al no ser la finalidad primera de esta memoria. La aplicación, aún siendo operativa, prosigue una mejora continua a medida que sugieren nuevas necesidades y que el sistema de toma de datos evoluciona.

En primer lugar, se ha debido de efecuar una primera elección respecto al lenguaje de programación que se ha utilizado. Teniendo en cuenta los requerimientos de este desarrollo, a saber, el ser lo mas flexible posible y el poder funcionar sobre un número de plataformas heterogéneas (intel, arm), se ha escogido un lenguaje sencillo pero que funcione de manera idéntica bajo varios sistemas operativos y bajo varias plataformas. Python posee estas características, es un lenguaje estructurado alrededor de la noción de objetos e incluye un amplio soporte. Además, posee extensiones para el manejo de tableros de datos (Numpy), su procesado (Scipy) y visualización (Matplotlib) que lo hacen muy adecuado para esta aplicación. Por último, permite desarrollar aplicaciones gráficas basándose en librerías estándar (Qt, GTK) gracias a herramientas ad-hoc (glade, qtcreator, etc.), incluyendo una interfaz hacia las librerías root (PyRoot), muy utilizadas en el entorno de la física nuclear y de partículas. Todas estas características han hecho que el lenguaje Python haya sido elegido como herramienta de desarrollo de nuestra aplicación.

Partiendo de esta elección, se ha procedido en primer lugar a importar la librería creada dentro del entorno python. Para ello se ha creado un módulo basándose en la utilidad *ctypes*. Esta utilidad da acceso a tipos compatibles con la librería c estándar, permitiendo crear envolventes alrededor de librerías genéricas. Su funcionamiento es sencillo, ya que es suficiente con cargar en memoria la librería y definir la interfaz de llamada hacia tipos de datos comprensibles por python. De esta manera se ha comenzado por crear un módulo que hará las veces de pasarela entre el nivel intermedio descrito anteriormente y la capa de aplicación. Esta última adopta en nuestro caso la forma de una clase Python (figura 7) que, en primera instancia, procede a importar el módulo anterior,

ejecutando llamadas a la librería de nivel intermedio presentada anteriormente de manera ordenada y estructurada.

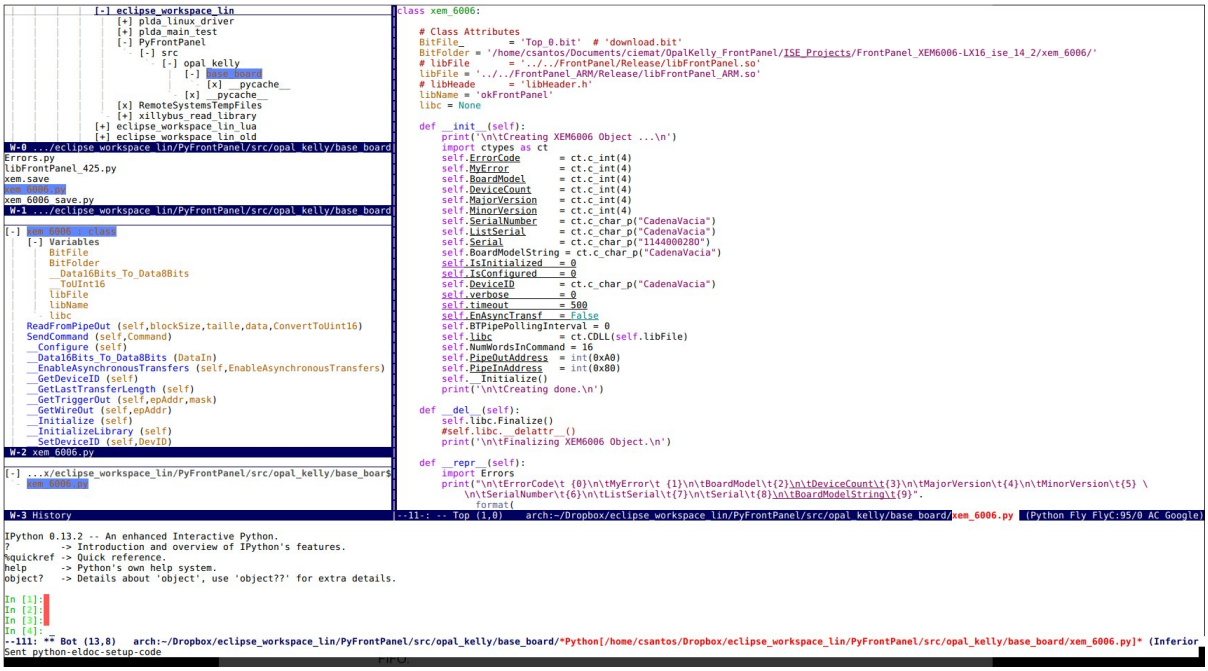


Figura 7: Ejemplo de capa de aplicación de nivel superior: desarrollo de una clase en lenguaje python

Esta forma de proceder simplifica y facilita el mantenimiento del software, clarificando su documentación y lisibilidad. La clase desarrollada posee una estructura organizada de creación, destrucción y manipulación de objetos que se identifica fácilmente con el uso del sistema de toma de datos. La aplicación se limitará a crear un objeto, ejecutando llamadas a los métodos del mismo, teniendo cada uno de ellos un significado evidente (configuración de los registros, recuperación de datos, etc.). Estas llamadas serán interpretadas desde el código embebido descrito en las secciones siguientes de esta memoria, en las que también se mostrarán varios ejemplos de aplicaciones de tests desarrolladas a partir del código expuesto aquí. Estas aplicaciones se limitarán a instanciar una referencia al sistema, con lo que el funcionamiento inferior y los detalles de implementación les serán transparentes, facilitando e independizando de esta manera las tareas de desarrollo a realizar, y respetando una metodología de desarrollo jerárquica y modular, que será útil a la hora de desarrollar aplicaciones diferentes reutilizando en gran medida parte del código ya existente.

Por último, es de destacar que la forma de proceder presentada aquí supone únicamente un ejemplo de tipo de desarrollo software, pudiéndose igualmente proceder a partir de otros lenguajes de alto nivel siguiendo metodologías distintas a partir de la librería de nivel intermedio presentada en la sección anterior.

2. Firmware

Hasta este momento el desarrollo de este proyecto se ha fundamentado en una concepción software basada en el uso de lenguajes de nivel alto (c, python). Este nivel de desarrollo implica una ejecución del código desarrollado en un entorno de tipo pc, para cualquier arquitectura, y se subdivide en un nivel explícito de comunicación y otro más elevado de nivel aplicativo. En ambos casos, se asumía una comunicación con un entorno embebido en el cual se ejecutaba un código encargado de gestionar todo el tráfico de datos así como la configuración de los módulos electrónicos.

En esta sección se describe en detalle la implementación del código embebido realizada para este proyecto. Como en el caso del nivel software, todo el desarrollo se basa en un requerimiento de flexibilidad y modularidad. Esto permitirá en un futuro una evolución, adaptación o modificación del código de manera ordenada y sistemática, con el objetivo de cubrir un rango de aplicaciones distintas de la inicialmente prevista.

En el desarrollo firmware se ha procedido por etapas, desde la mas sencilla (comunicación de bajo nivel) hasta la más compleja (toma de datos, procesado, etc.). De esta manera se ha comenzado por implementar un diseño vhdl que reutilice parte de la librería hardware FP, estudiando su comportamiento. Posteriormente se ha procedido a incorporar este diseño a una arquitectura de nivel superior basada en una lógica de procesador embebido, que permita el desarrollo de aplicaciones de forma mas abierta y flexible. Por último, lo anterior se ha completado con bloques periféricos a dicho procesador que faciliten la creación de rutinas de acceso a los recursos existentes en el sistema de toma de datos, permitiendo su uso de manera simplificada.

2.1. Comunicación de bajo nivel I

Se ha comenzado por realizar una primera implementación firmware con el objetivo de adquirir el conocimiento necesario que permita desarrollos posteriores más complejos. Este primer diseño tiene por objetivo único establecer una comunicación rudimentaria entre el software anteriormente explicado y el módulo hardware Shuttle LX1 a partir de un subconjunto de los elementos contenidos en la librería FrontPanel. De esta manera, se establecerán y entenderán los elementos de base que serán utilizados posteriormente.

Desde una perspectiva de diseño hardware en vhdl, esta biblioteca viene dada como una serie de ficheros de netlist *.ngc con cada uno de los elementos que la componen: un controlador (*okHost*), un or lógico (*okWireOR*) y un componente por endpoint (*wires*, *pipes*, *triggers*, etc.), como se muestra en la figura 8. En nuestro caso tan sólo nos interesarán los endpoints de tipo *okBTP*. La definición de cada uno de estos componentes se da en el fichero *okLibrary.v(hd)* para su instanciación en nuestro diseño.

Se comentó anteriormente que los recursos lógicos existentes dentro de la FPGA son limitados. Es por ello que estos recursos han de ser reservados para tareas de gestión y procesado de datos, minimizando cualquier otro uso que de ellos pueda hacerse. Por ejemplo, se desea reducir al máximo los recursos ocupados por la implementación de la comunicación usb. El bloque controlador es necesario en cualquier caso, no así los demás bloques. Es por ello que el único tipo de endpoint relevante para esta aplicación será el llamado BTP, que permite funcionar enviando / recibiendo un tablero de hasta 1024 bytes desde / hacia el colector de datos de manera síncrona a un reloj de 48 MHz, haciéndolo además a una velocidad de transferencia máxima (que dependerá del tamaño de bloque de datos solicitado). Esto se logra obviamente para tamaños de paquetes de bytes grandes, lo que minimiza el envío de informaciones de control a nivel de la API y del sistema operativo así como la sobrecarga implícita en cada llamada.

Así, en este diseño, tan solo se han instanciado tres bloques: el controlador, un BTP de entrada

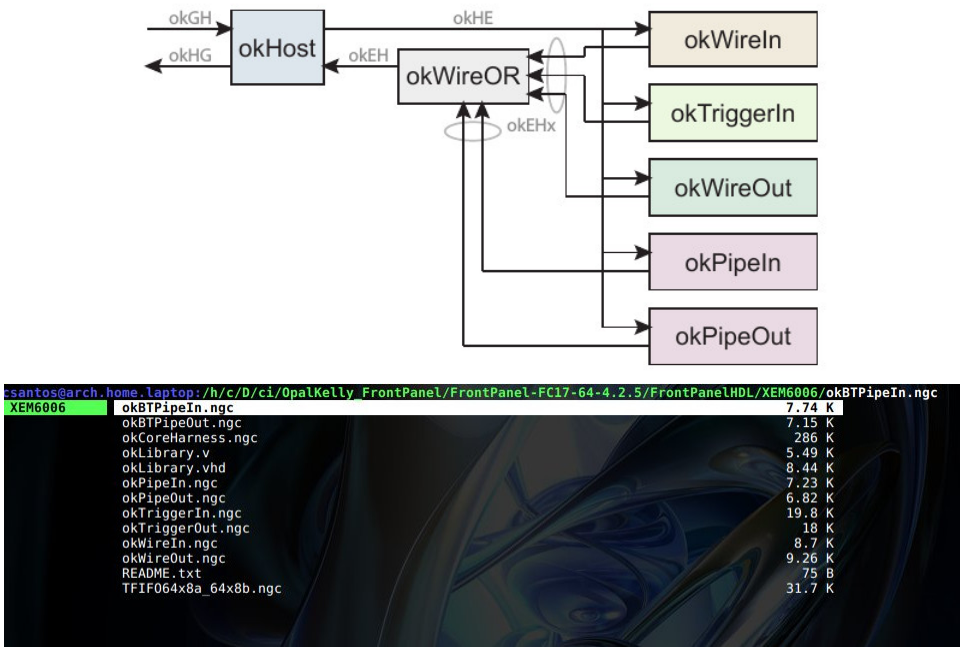


Figura 8: Endpoints y elementos hardware de la librería FrontPanel

a la FPGA para el envío de comandos, de tamaño reducido, y un BTP de salida de la FPGA para el envío de datos hacia el pc colector. Los endpoints son siempre maestros en la comunicación, teniendo acceso únicamente a una señal de RDY posicionada desde el colector por software.

A partir de estos elementos se ha desarrollado un código vhdl (figura 9) que permite el envío de informaciones hacia / desde el módulo. En primer lugar, se ha de definir una interfaz estándar del nivel top de nuestro diseño: un reloj de entrada de 48 Mhz (*sys-clk*) sincroniza todo el diseño y permite enviar datos sobre un bus de 8 bits (*hi-in*), además de actuar sobre cuatro leds. Las demás señales se consideran señales de control del protocolo empleado. Se define también un genérico que fije el tamaño del banco de registros donde almacenar los datos transferidos.

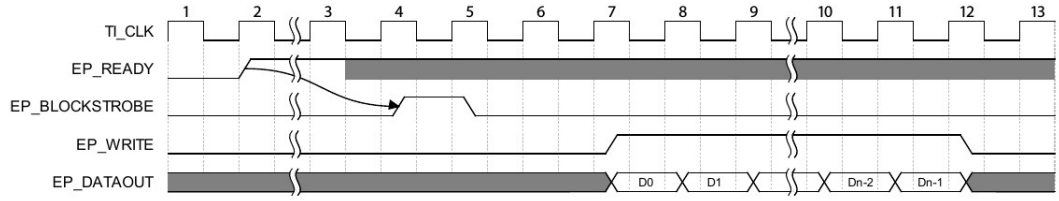
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

library UNISIM;
use UNISIM.VComponents.all;
use work.FRONTPANEL.all;

entity Top_0 is
  generic (RegBankLength : natural := 3);
  port (sys_clk  : in    std_logic;
        hi_in    : in    std_logic_vector(7 downto 0);
        hi_out   : out   std_logic_vector(1 downto 0);
        hi_inout : inout std_logic_vector(15 downto 0);
        hi_aa    : inout std_logic;
        led      : out   std_logic_vector(3 downto 0));
end Top_0;

architecture Behavioral of Top_0 is

  signal ti_clk, ep_out_read                    : std_logic
  signal ok1                                    : std_logic_vector(30 downto 0)
  signal ok2                                    : std_logic_vector(16 downto 0)
  signal ok2s                                   : std_logic_vector(17*2-1 downto 0)
  type RegBank is array(0 to (2**RegBankLength-1)) of std_logic_vector(15 downto 0);
  signal my_RegBank                             : RegBank;
  signal index_in, index_out                    : unsigned(RegBankLength-1 downto 0)
  signal up_counter                             : unsigned(15 downto 0)
  signal ep_in_blockstrobe, ep_in_write         : std_logic
  signal ep_in_dataout, ep_out_datain           : std_logic_vector(15 downto 0)
  signal ep_out_ready, ep_out_strobe, interrupt : std_logic
  signal led_tmp                                : std_logic_vector(3 downto 0)
  signal time_counter, time_counter2            : unsigned(31 downto 0)

begin
```

```
begin

  okWO : okWireOR
    generic map (N => 2)
    port map (ok2  => ok2,              -- out, 17b
              ok2s => ok2s);            -- in,  34b = 17*2-1

  okHI : okHost
    port map (hi_in    => hi_in,
              hi_out   => hi_out,
              hi_inout => hi_inout,
              hi_aa    => hi_aa,
              ti_clk   => ti_clk,
              ok1      => ok1,          -- out, 31b
              ok2      => ok2);         -- in

  pipeIn9C : okBTPipeIn
    port map (ok1           => ok1,     -- in, 31b
              ok2           => ok2s(1*17-1 downto 0*17),  --out, 34b, rightmost bits
              ep_addr       => X"80",
              ep_dataout    => ep_in_dataout,
              ep_write      => ep_in_write,
              ep_blockstrobe => ep_in_blockstrobe,
              ep_ready      => '1');

  pipeOutA3 : okBTPipeOut
    port map (ok1           => ok1,                       -- in, 31b
              ok2           => ok2s(2*17-1 downto 1*17),  -- out, 34b,
                                                          -- leftmost bits
              ep_addr       => X"A0",
              ep_datain     => ep_out_datain,
              ep_read       => ep_out_read,
              ep_blockstrobe => ep_out_strobe,
              ep_ready      => ep_out_ready);
```

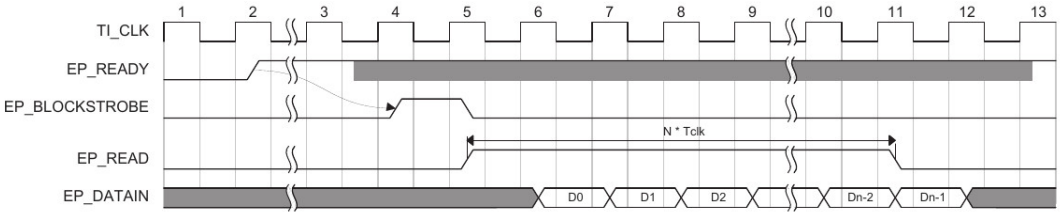
Figura 9: Implementación de los módulos BTP de la librería FrontPanel

Con el propósito de reducir los recursos utilizados por el protocolo de comunicación, este diseño contendrá únicamente un controlador y dos elementos de envío hacia la FPGA (*okBTPIn*) y desde la FPGA (*okBTPOut*). Se puede apreciar como el controlador comunica a través del puerto *ok1* de 31 bits con los endpoints BTP, que a su vez devuelven una señal *oks2*. Las dos mitades de ésta última se combinan dentro del bloque *okWireOR* y se devuelven como *ok2* al bloque controlador. Además, éste provee una señal *ti-clk*, que se ha verificado es idéntica a la señal *sys-clk*.

A partir de estos elementos de base puede comenzar a considerarse la problemática de la transferencia de datos. Los dos bloques (*okBTP*) son implícitamente síncronos con el reloj (*ti-clk*). El bloque (*okBTPIn*) posee un interfaz de recepcióin de datos en el cual se define una dirección arbitraria para poder referenciarlo desde el software. Esta interfaz se corresponde al cronograma mostrado en la figura 10-a. El software es maestro en la comunicación, pero el hardware habilita una señal *ep-ready* habilitando la transmisión de una trama completa, de modo que el diseño debe ser capaz de alamacenar estos datos. Este tipo de transmisión se adapta tipicamente a un modelo de diseño en el que una memoria (o bien fifo) posiciona un flag cuando un número suficiente de posiciones estan disponibles, almacenando muestras *ep-dataout* de entrada a partir de una señal de escritura *ep-write*. Adicionalmente, la señal *ep-blockstrobe* permite verificar al diseño el inicio de la transición. Este simple modelo permite una transmisión de informaciones desde una aplicación software de control.



(a) Entrada



(b) Salida

Figura 10: Cronograma de tipo de transmisión Block-Throttled de entrada / salida

El bloque (*okBTPOut*) se encarga del envío de informaciones en salida del módulo hardware. En este caso, la aplicación de gestión desde el lado del pc continúa siendo maestra de la comunicación. Así, cuando la señal *ep-read* pasa a uno en un ciclo de reloj dado, el protocolo de salida transmitirá el dato presente en *ep-datain* en el ciclo de reloj siguiente, siendo la responsabilidad del diseño el gestionar adecuadamente este hecho. Para ello, el diseño tiene la posibilidad de actuar sobre la señal *ep-ready* para indicar que un bloque completo de datos está disponible para su envío. De nuevo, la señal *ep-blockstrobe* da una indicación sobre el inicio de la transmisión de datos. Este cronograma se resume en la figura 10-b. De lo anterior, puede entenderse que este protocolo se corresponde con un modelo sencillo donde el diseño deposita datos de salida en un elemento de memoria temporal (fifo), siendo el propio *okBTPOut* quien se ocupe de su repatriación hacia la aplicación de colección de datos, incluyendo la latencia de un ciclo de reloj característica para este tipo de elemento de memoria. Como se puede ver, los bloques BTP utilizan un mínimo de recursos, que serán esencialmente iguales

al tamaño de las memorias de intercambio de datos en los dos sentidos. El tamaño la memoria en el sentido controlador hacia sistema de toma de datos se reduce al máximo al tener una dimensión igual a 16 bits veces el número de registros enviados (16 en este proyecto, pues no se pretende transferir cantidades elevadas de datos hacia el sistema), implementándose a partir de elementos de memoria distribuida, lo que libera los bloques de memoria embebidos. En el sentido contrario, se utilizarán bloques de memoria RAM embebidos: a mayor tamaño se obtiene una velocidad de transferencia de datos también superior. Este punto se verá detalle mas adelante, al construirse una interfaz basada en el uso de buses FSL.

```
-- interrupt and register bank update
process (ti_clk)
begin
  if (ti_clk = '1' and ti_clk'event) then
    --
    if ep_in_blockstrobe = '1' then
      index_in <= (others => '0');
    elsif ep_in_write = '1' then
      my_RegBank(to_integer(index_in)) <= ep_in_dataout;
      index_in                         <= index_in + 1;
    end if;
    -- interrupt
    if ep_in_blockstrobe = '1' then
      interrupt <= '1';
    elsif index_in = 2**RegBankLength-1 then
      interrupt <= '0';
    end if;
    --
    if (index_in = 2**RegBankLength-1 and (my_RegBank(0) = X"0001" or my_RegBank(0) = X"0002")
      ep_out_ready <= '1';
    elsif ep_out_strobe = '1' and my_RegBank(0) = X"0001" then
      ep_out_ready <= '0';
    end if;
  end if;
end process;

--
process (ti_clk)
begin
  if (ti_clk = '1' and ti_clk'event) then
    --
    if interrupt = '1' then
      index_out  <= (others => '0');
      up_counter <= (others => '0');
    elsif ep_out_read = '1' then
      if my_RegBank(0) = X"0001" then
        ep_out_datain <= my_RegBank(to_integer(index_out));
        index_out     <= index_out + 1;
      elsif my_RegBank(0) = X"0002" then
        --ep_out_datain <= std_logic_vector(up_counter(7 downto 0)&up_counter(15 downto 8));
        ep_out_datain <= std_logic_vector(up_counter);
        up_counter    <= up_counter + 1;
      else
        ep_out_datain <= X"ABCD";
      end if;
    end if;
  end if;
end process;
```

```
led <= not(led_tmp);

process (ti_clk)
begin
  if (ti_clk = '1' and ti_clk'event) then
    if time_counter(29) = '1' then
      time_counter        <= (others => '0');
      led_tmp(1 downto 0) <= not(led_tmp(1 downto 0));
    else
      time_counter <= time_counter + 1;
    end if;
  end if;
end process;

process (sys_clk)
begin
  if (sys_clk = '1' and sys_clk'event) then
    if time_counter2(29) = '1' then
      time_counter2       <= (others => '0');
      led_tmp(3 downto 2) <= not(led_tmp(3 downto 2));
    else
      time_counter2 <= time_counter2 + 1;
    end if;
  end if;
end process;
```

Figura 11: Código de recepción / transmisión de datos

Teniendo en cuenta las consideraciones anteriores, la lógica que siga a los dos bloques endpoint *BTP* deberá jugar el papel de interfaz entre éstos y toda la funcionalidad que se desee implementar a continuación. En este primer ejemplo se ha optado por crear un array *My-RegBank* de tamaño dado por el genérico definido en la interfaz de nivel superior al diseño, y de ancho de palabra de 16 bits. En este caso sencillo, el bloque de entrada aparece habilitado en continuo para recibir datos, que se almacenan en este array de manera secuencial. La señal *index-in* servirá de índice de escritura en este array: al iniciarse una transmisión de datos, la señal de strobe permite resetear el valor de este índice, de modo que para cada nueva transferencia, el primer dato se almacena en la posición primera del array. En paralelo a la lógica anterior se produce una señal de *interrupt* únicamente durante la transferencia de datos. Por último, dependiendo del valor almacenado en la primera posición del array, se posiciona el valor del registro *ep-out-ready*. Este código se muestra en la figura 11.

La lógica anterior implementa el almacenamiento de informaciones recibidas desde la aplicación software. Un segundo bloque de lógica implementa el reenvío de datos en el otro sentido a través del endpoint de salida. La señal de interrupción anterior provoca un reset de este bloque. Al final del mismo, un nivel alto de *ep-out-read* activa el envío de datos. Estos datos pueden ser bien el contenido del array de almacenamiento de datos de entrada, bien el valor de un contador incrementado a cada ciclo de reloj, bien un valor fijo. En el primer caso, el envío de datos se detiene al finalizar de recorrer

el array, no así en los demas casos.

Si tenemos en cuenta que el reenvío de datos (acción) se activa en función del contenido de los registros de entrada recibidos (comandos), ya tenemos un protocolo de control y salida de datos desde el módulo hardware hacia el módulo controlador software. Este código mínimo supondrá a partir de este momento la base de desarrollos futuros. En secciones posteriores de esta memoria se presentarán algunos resultados respecto al rendimiento en términos de velocidad de transferencia de datos así como algunos test de tasa de errores de transmisión que permitirán validar esta implementación.

2.2. Comunicación de bajo nivel II

```
-- interrupt and register bank update
process (ti_clk)
begin
  if (ti_clk = '1' and ti_clk'event) then
    --
    if ep_in_blockstrobe = '1' then
      index_in <= (others => '0');
    elsif ep_in_write = '1' then
      my_RegBank(to_integer(index_in)) <= ep_in_dataout;
      index_in                         <= index_in + 1;
    end if;
    -- interrupt
    if ep_in_blockstrobe = '1' then
      interrupt <= '1';
    elsif index_in = 2**RegBankLength-1 then
      interrupt <= '0';
    end if;
    --
    if (index_in = 2**RegBankLength-1 and (my_RegBank(0) = X"0001" or my_RegBank(0) = X"00
      ep_out_ready <= '1';
    elsif ep_out_strobe = '1' and my_RegBank(0) = X"0001" then
      ep_out_ready <= '0';
    elsif my_RegBank(0) = X"0003" then
      ep_out_ready <= storage_full;
    end if;
  end if;
end process;

--
process (ti_clk)
begin
  if (ti_clk = '1' and ti_clk'event) then
    --
    if interrupt = '1' then
      index_out     <= (others => '0');
      up_counter    <= (others => '0');
      storage_wr_en <= '0';
    elsif ep_out_read = '1' then
      storage_wr_en <= '0';
      if my_RegBank(0) = X"0001" then
        ep_out_datain <= my_RegBank(to_integer(index_out));
        index_out     <= index_out + 1;
      elsif my_RegBank(0) = X"0002" then
        ep_out_datain <= std_logic_vector(up_counter);
        up_counter    <= up_counter + 1;
      elsif my_RegBank(0) = X"0003" then
        ep_out_datain <= storage_dout;
      else
        ep_out_datain <= X"ABCD";
      end if;
    elsif my_RegBank(0) = X"0003" then
      storage_wr_en <= '1';
      up_counter    <= up_counter + 1;
    end if;
  end if;
end process;
```

```
process (ti_clk)
begin
  if (ti_clk = '1' and ti_clk'event) then
    --
    if interrupt = '1' then
      index_out     <= (others => '0');
      up_counter    <= (others => '0');
      storage_wr_en <= '0';
    elsif ep_out_read = '1' then
      storage_wr_en <= '0';
      if my_RegBank(0) = X"0001" then
        ep_out_datain <= my_RegBank(to_integer(index_out));
        index_out     <= index_out + 1;
      elsif my_RegBank(0) = X"0002" then
        ep_out_datain <= std_logic_vector(up_counter);
        up_counter    <= up_counter + 1;
      elsif my_RegBank(0) = X"0003" then
        ep_out_datain <= storage_dout;
      else
        ep_out_datain <= X"ABCD";
      end if;
    elsif my_RegBank(0) = X"0003" then
      storage_wr_en <= '1';
      up_counter    <= up_counter + 1;
    end if;
  end if;
end process;

led_tmp <= X"1" when my_RegBank(0) = X"0001" else
           X"2" when my_RegBank(0) = X"0002" else
           X"3" when my_RegBank(0) = X"0003" else
           X"0";

led <= not(led_tmp);

--led <= not(my_RegBank(0)(3 downto 0));

------------------------------------------------------------------
--
------------------------------------------------------------------
storage_rst   <= interrupt;
storage_rd_en <= ep_out_read;
storage_din   <= std_logic_vector(up_counter);

-- in first word fall through mode
u_storage : storage
  port map (clk   => ti_clk,
            srst  => storage_rst,
            din   => storage_din,
            wr_en => storage_wr_en,
            rd_en => storage_rd_en,
            dout  => storage_dout,
            full  => storage_full,
            empty => storage_empty);
```

Figura 12: Código de recepción / transmisión de datos por bloques

Siguiendo con el protocolo implementado en la sección anterior, se ha procedido a continuación a remplazar el array de almacenamiento de datos por un espacio de almacenamiento mayor (fifo) de hasta 1024 posiciones. Esta mejora persigue un doble objetivo. Por un lado, se pretende estudiar el comportamiento del sistema para transferencias de datos mayores (en ambos sentidos), y por otra parte se busca emular el comportamiento final del firmware, donde los datos en salida se almacenarán en una fifo de este mismo tamaño que será igual al máximo tamaño de bloque permitido por las librerías FrontPanel para un rendimiento óptimo cercano a los 30 MB/s.

El código referido en la figura 12 muestra el código correspondiente, siendo similar al anterior. En esta ocasión un comando predefinido habilita la salida de datos mediante la aserción de la señal *ep-out-ready*, que ha sido conectada al flag de lleno completo de la fifo de almacenamiento. Esta fifo se llena automáticamente con un valor arbitrario iguala un contador. La secuencia seguida es, en primer lugar, llenado completo de la fifo; a continuación, el flag de llenado completo (*storage-full*)

pasa a nivel alto. Esta información se transmite al driver mediante el flag *ep-out-ready*, que solicita el envío de datos a través de la señal *ep-out-read*, que lee el contenido de la fifo y cuya salida se sitúa en el puerto de salida de datos *ep-out-datain*.

2.3. Tests funcionales

Con el objetivo de validar y caracterizar la lógica de bajo nivel desarrollada hasta este punto, se ha procedido a implementar una serie de baterías de tests a partir del código software descrito anteriormente. Estos tests tienen por finalidad el comprobar que el protocolo de comunicación y transferencia de datos establecido hasta este punto es adecuado y carece de errores de transmisión que podrían surgir al someter al sistema a condiciones límites.

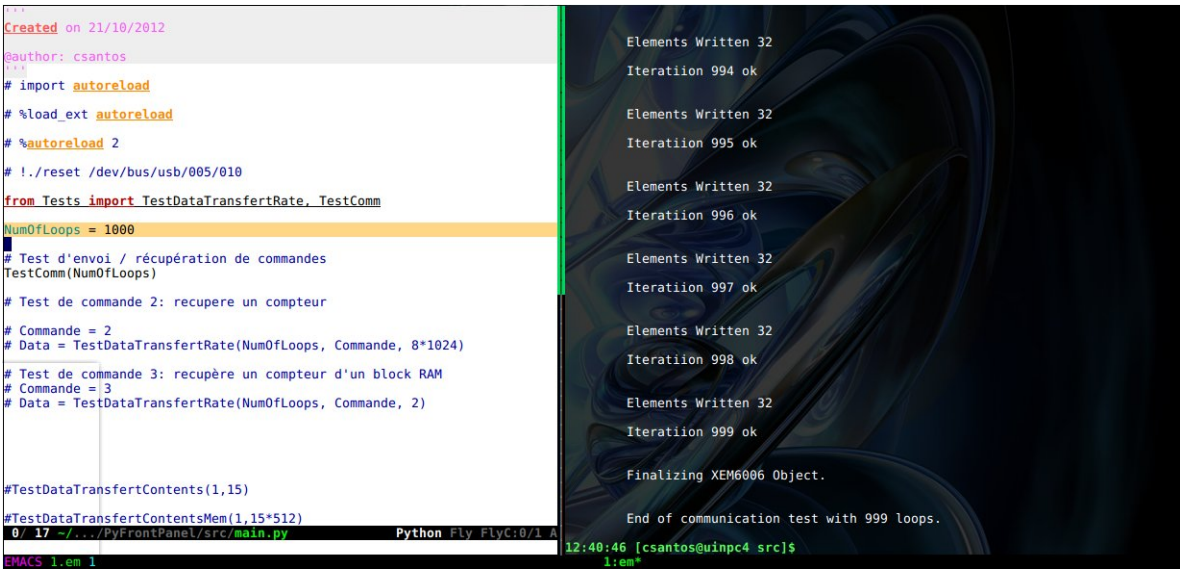


Figura 13: Tests de transmisión / recepción de comandos

En primer lugar se ha comprobado como la transmisión de los registros de control hacia el sistema no produce errores (figura 13). Para ello, se ha procedido a generar un banco de valores aleatorios. Estos valores se transmiten al sistema, que al decodificar el comando 0x01 como valor inicial de este banco procede a reenviar los datos recibidos. Este test persigue una finalidad doble, por un lado, se verifica que los datos recibidos se alinean correctamente, lo que es de crucial importancia al ser el valor decodificado en primera posición el que indique la acción que deba de ejecutar el sistema. Por otra parte, se verifica que los valores almacenados se corresponden con los transmitidos. Además, al iterarse el test sobre un número elevado de ciclos se comprueba que no se producen errores bajo condiciones reales de operación.

En segundo lugar, se ha desarrollado un segundo test consistente en el reenvío masivo de datos hacia el colector de datos (figura 14). De nuevo, se pretende aquí asegurarse de que un elemento clave del sistema de toma de datos, como es la salida de datos a alta tasa, no produce errores. Para ello se procede enviando el comando 0x02, que sera decodificado por la lógica dentro de la FPGA e interpretado como una solicitud de reenvío de datos continuada. Estos datos poseen un patrón conocido, un contador incremental de una unidad. Esto permite desde el pc colector medir tanto la tasa de transferencia (alrededor de 30 MB/s.) como la existencia de errores de transmisión, que no han sido observados.

Por ultimo, y con la finalidad de acercarse lo mas posible a una simulación de la arquitectura final del sistema de toma de datos, se ha desarrollado una variante del tests anterior, en el que el

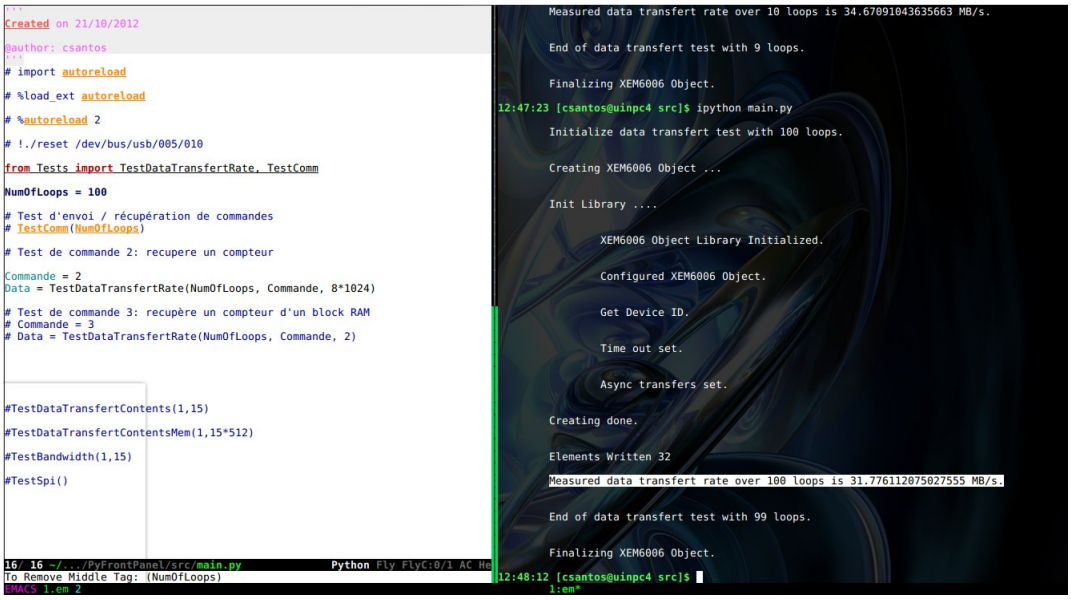


Figura 14: Tests de transmisión / recepción de datos

contador incremental rellena una memoria FIFO intermedia. Una vez esta memoria llena, la lógica de reenvío de datos se activa. Este proceso se repite iterativamente durante un número de ciclos como en los casos anteriores, observándose nuevamente una transmisión correcta de datos.

3. Comunicación a través de un procesador

Todo el desarrollo firmware efectuado hasta este punto se ha centrado en el establecimiento de una comunicación fiable, ligera (en el sentido de recurso lógicos empleados) y de alto rendimiento entre los módulos colector y procesador del sistema de toma de datos. Esta comunicación se fundamenta en el almacenamiento de las informaciones a intercambiar entre estos dos módulos en una memoria fifo intermedia. Esta fifo actuará como frontera entre el bus de datos usb y la lógica propia a la aplicación que desee implementarse. Hasta este momento, dicha lógica consiste únicamente en la generación de datos de prueba que permitan caracterizar la comunicación de forma adecuada. Todo este desarrollo ha sido efectuado mediante la escritura de código vhdl de nivel bajo, lo que no es el objetivo final de este proyecto, sino tan solo una etapa intermedia.

A partir de este punto se procederá a efectuar un salto cualitativo de implementación firmware. Siendo uno de los requerimientos fundamentales de este sistema el ser lo más flexible posible, la escritura únicamente de código vhdl resultaría ser un factor penalizante. Esto es así al consistir en una herramienta de desarrollo poco accesible por un usuario que desee adaptar el sistema a una aplicación distinta de la originalmente ideada. Es por ello que se ha procedido a una reescritura parcial del protocolo de comunicación partiendo de un punto de vista sistema de nivel superior, más genérico y por lo tanto mas accesible.

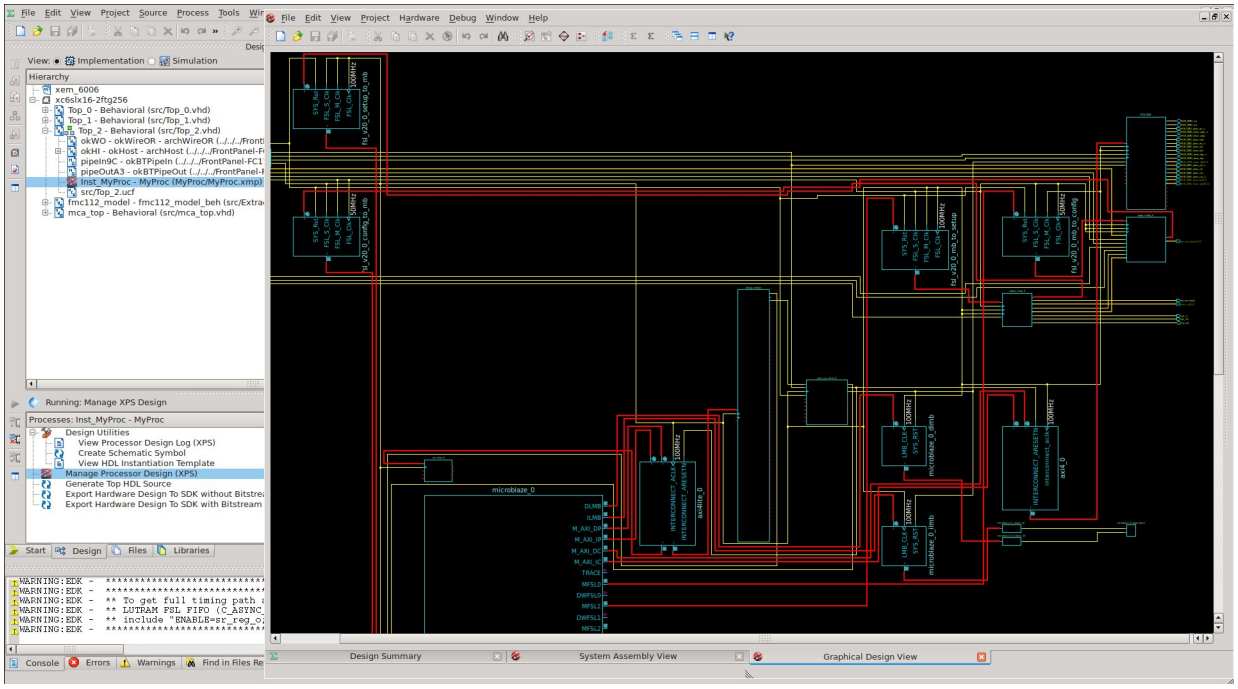


Figura 15: Plataforma procesador de gestión

Por todo lo anterior, se ha decidido el implementar un procesador software en el interior de la fpga del módulo procesador con un objetivo doble. Por un lado, este procesador permite el desarrollo de firmware estructurado y modulable al que se podrá acceder mediante código c, mas accesible a un usuario en el caso de desear desarrollar una aplicación sobre esta plataforma. Por otra parte, se simplifica la gestión del módulo de muestreo de señales, que ha se ser configurado y gestionado de manera que se puedan recuperar las muestras de entrada que serán, a continuación, tratadas. Este procesador ha sido desarrollado a partir de la herramienta EDK de Xilinx, como puede observarse

en la figura 15, donde se observa el procesador central junto a los periféricos que lo completan, y que serán descritos en la secciones siguientes de esta memoria.

3.1. Creación de la plataforma

El primer objetivo de esta parte del desarrollo consiste en portar el código desarrollado hasta este punto adaptándolo a una implementación basada en un procesador. Para ello se ha seguido una metodología similar a la anterior, esto es, se ha privilegiado una concepción modular en la que se minimizen los recursos lógicos utilizados. De nuevo, se pretende aquí que la lógica de transferencia de datos ocupe un espacio mínimo dentro de la FPGA. selección requisito condiciona la elección de arquitectura realizada y que sera detallada a continuación.

En primer lugar, la concepción llevada a acabo se ha basado en el uso de buses de tipo FSL (Fast Simplex Link) entre el procesador MicroBlaze (uB) y los bloques de lógica de propósito específico desarrollados. Esta selección está motivada por el hecho de que dicho bus permite una concepción de bajo nivel basada el en intercambio de datos a través de una memoria FIFO configurable. Además, permite un control absoluto sobre la lógica resultante, que será sencilla y flexible (gestión de asincronismo entre dominios de relojes, recursos de tipo SRL o BlockRAM, tamaño variable, etc.).uB permite el uso de un número elevado de buses FSL, que además simplifican la concepción de software al poderse acceder al bus por medio de instrucciones simples de tipo put / get. Este punto es importante si se tiene en mente el requerimiento de permitir a un usuario cualquiera el poder desarrollar su propio código sobre esta arquitectura.

De esta forma se ha desarrollado un primer bloque funcional (figura 16) de intercambio de datos entre la biblioteca FrontPanel y uB. Este bloque tiene por objetivo el de sintetizar un interfaz entre la lógica vista en las secciones anteriores de esta memoria y una lógica de tipo FIFO. Incluye una trasición entre los dominios de reloj de 48 MHz (usb) y 100 MHz (uB). Además, permite transmitir datos hacia el procesador procedentes tanto de los bloques FP como de una segunda fuente adicional. Según el origen de los datos, añadirá una etiqueta que permitirá su tratamiento posterior por software.

```
-- From USB to uB, clock domain 50 MHz
USB_to_uB: process (FSL_M_Clk)
begin
  if (FSL_M_Clk'event and FSL_M_Clk = '1') then
    if ep_out_strobe = '1' then
      FSL_M_Data <= (others => '0');
    elsif ep_in_write = '1' then
      FSL_M_Data <= X"0001" & ep_in_dataout;
    elsif enable_write = '1' then
      FSL_M_Data <= X"0002" & data_write;
    else
      FSL_M_Data <= X"ABCD_ABCD";
    end if;
    FSL_M_Write   <= ep_in_write or ep_out_strobe or enable_write;
    FSL_M_Control <= '0';
  end if;
end process;

-- from uB to USB, clock domain 50 MHz
uB_to_USB: process (FSL_S_Clk)
begin
  if (FSL_S_Clk'event and FSL_S_Clk = '1') then
    if ep_out_read = '1' then
      un_sur_deux_out <= not(un_sur_deux_out);
    end if;
    if un_sur_deux_out = '1' then
      ep_out_datain <= FSL_S_Data(16 to 31);
    else
      ep_out_datain <= FSL_S_Data(0 to 15);
    end if;
  --FSL_S_Read <= ep_out_read and un_sur_deux_out;
  end if;
end process;

--ep_out_datain <= FSL_S_Data(16 to 31) when un_sur_deux_out = '1' else FSL_S_Data(0 to 15);
FSL_S_Read <= ep_out_read and un_sur_deux_out;
```

Figura 16: Periférico de interfaz entre FP y uB

Hay que tener en cuenta que esta aplicación requiere únicamente el envío de 16 registros de con-

figuración desde el pc de control, siendo el primero de ellos el comando a ejecutar y los restantes los parámetro de dicho comando. Es por ello que la FIFO asíncrona de entrada a uB tendrá únicamente 16 posiciones y será sintetizada a partir de lógica distribuida, ahorrando recursos. La existencia de datos en esta FIFO provocará una interrupción en uB, que deberá saber gestionar desde el vector de interrupciones. Esta interrupción será única, con el objetivo de no requerir un controlador de interrupciones. Así, la señal de "ep-out-strobe", indicadora de inicio del envío de datos desde el pc generará un valor nulo, que podrá ser utilizado desde la rutina de interrupción al ser el primer valor obtenido en cada interrupción. Esta rutina deberá de almacenar el resto de valores transmitidos desde el pc, actuando posteriormente según los valores transmitidos.

Finalmente, este primer bloque accede a los datos disponibles en salida del procesador uB, transmitiéndolos hacia el usb y gestionando el tránsito entre los buses de 32 bits (uB) y 16 bits (usb). En este caso, la memoria FIFO asíncrona intermedia será sintetizada a partir de bloques de memoria BlockRAM de una profundidad de 4096 posiciones con el objetivo de maximizar el rendimiento en la transferencia de datos. La figura 16 ilustra parte de la lógica, que pretende ser lo más ligera posible.

La lógica anterior permite un tránsito de datos desde / hacia el colector de datos (pc). Sin embargo, se requiere una lógica adicional que permita acceder a los recursos restantes del sistema de toma de datos desde el procesador embebido, además de poder señalar al pc colector la presencia de datos en salida (señal "ep-out-rdy"). Con este objetivo, se ha desarrollado un segundo bloque periférico conectado a un segundo bus FSL (figura 17). En esta ocasión, la lógica desarrollada permite recuperar los datos depositados por uB en una FIFO asíncrona de 16 posiciones implementada con ayuda de recursos distribuidos de manera similar al caso anterior. En esta ocasión, uB actúa como procesador de configuración de periféricos.

La lógica existente en este segundo bloque decodifica el valor transmitido y actúa sobre los periféricos. Se puede actuar sobre los leds con el objetivo de verificar el diseño visualmente, además de transmitir al pc colector la existencia de datos de la adquisición. Además, se pueden generar datos aleatorios que tendrán por objetivo el caracterizar el sistema, permitiendo dimensionarlo correctamente. Estos datos se introducirán en el bloque anterior y actuarán como segunda fuente de datos de entrada. Se puede igualmente configurar el periférico mas importante del sistema de toma de datos, que es la tarjeta mezzanine de muestreo de señales, como se verá mas adelante. Todas estas acciones pueden corresponder a instrucciones muy sencillas que se incluirán en el código software de uB. Un primer ejemplo de este software se detalla en la sección siguiente.

3.2. Rutinas software sobre microBlaze

Los bloques de lógica descritos en la sección anterior permiten la entrada y salida de datos desde uB. Como complemento, se ha desarrollado las rutinas software en código c que permiten acceder a dichos bloques desde el procesador de una forma sencilla para un usuario del sistema. Este código se muestra en la parte izquierda de la figura 17 y se describe en esta sección.

La estructura de código que se ha elegido consiste en implementar un bucle while infinito que no ejecute ninguna acción por defecto. Previamente se han configurado la cache de instrucciones y se han posicionado los leds de la tarjeta. El código permanece inactivo hasta el momento en que una interrupción externa se produce. Como se comentó anteriormente la única fuente de interrupciones proviene de la presencia de valores en la FIFO conectada a la entrada del canal usb. Estos datos se consideran como registros de configuración, y se han de almacenar localmente en un array "reg-bank" de 16 posiciones de tipo Xuin16.

Cuando la interrupción se produce, el vector de interrupción entra en juego. En primer lugar se recupera un valor inicial que debe de ser igual a 0, por construcción del bloque de interfaz entre FP

```
int main()
{
  microblaze_register_handler(timer_int_handler, 0);
  microblaze_enable_interrupts();

  /* Typically, before using the cache, your program must perform a particular sequence of cache
     operations to ensure that invalid/dirty data in the cache is not being used by the processor. This
     would typically happen during repeated program downloads and executions.
     Initialize ICache */

  microblaze_invalidate_icache();
  microblaze_enable_icache ();

  putfslx(0x00010000 | 0x0003, 1, FSL_DEFAULT);

  LedsValue = 0;
  ExecuteAction = 0;
  UpdateOutputs

    while(1){
      /*
        ExecuteAction goes high only right after and after interrupt
        meaning that an action must be executed once
      */
      if (ExecuteAction == 1){

        // puts this register in the leds
        putfslx(0x00010000 | reg_bank[0], 1, FSL_DEFAULT);

        // action following the value of reg_bank[0] (command)
        switch(reg_bank[0]){
        case 0:
          LedsValue = 0x00;
          break;
        case 1:
          LedsValue = 0x01;
          readback_reg_bank();
          break;
        case 2:
          LedsValue = 0x02;
          send_dummy_counter();
          break;
        case 3:
          LedsValue = 0x03;
          send_dummy_packet();
          break;
        case 4:
          LedsValue = 0x04;
          send_dummy_packet_mem();
          break;
        case 5:
          LedsValue = 0x05;
          test_bandwith();
          break;
        default:
          LedsValue = 0x06;
          break;
        } // switch
      } // if
    } // while


  microblaze_invalidate_dcache();
  microblaze_disable_dcache();
```

```
-------------------------------------------------------------------------------
-- 50 MHz clock domain
-------------------------------------------------------------------------------
SampleIn: process (FSL_S_Clk) is
begin
  if FSL_S_Clk'event and FSL_S_Clk = '1' then
    if FSL_Rst = '1' then
      FSL_S_Read <= '0';
      data_read  <= (others => '0');
    elsif FSL_S_Exists = '1' then
      data_read  <= FSL_S_Data;
      FSL_S_Read <= '1';
    else
      FSL_S_Read <= '0';
    end if;
  end if;
end process;

ProcesSample: process (FSL_S_Clk) is
begin
  if FSL_S_Clk'event and FSL_S_Clk = '1' then
    case data_read(0 to 15) is
      when X"0001" =>
        leds_out           <= data_read(28 to 31);
        ep_out_ready       <= data_read(27);
        enable_dummy_data_50 <= data_read(26);
        enable_spi         <= data_read(25);
      when X"0002" =>
        spi_length <= data_read(26 to 31);
      when X"0003" =>
        spi_data_in(0 to 15) <= data_read(16 to 31);
      when X"0004" =>
        spi_data_in(16 to 31) <= data_read(16 to 31);
      when others =>
        null;
    end case;
  end if;
end process;

process (FSL_S_Clk) is
begin
  if FSL_S_Clk'event and FSL_S_Clk = '1' then
    -- arret et reset au repos
    if enable_spi = '0' then
      spi_counter  <= (others => '1');
      spi_dio_o    <= '0';
      spi_dio_t    <= '1';
      spi_disable  <= '0';
      data_write   <= (others => '0');
      enable_write <= '0';
      spi_data_out <= (others => '0');
    else
      if spi_disable = '0' then
        spi_counter <= spi_counter + 1;
        -- condition d'arret
        if unsigned(spi_cycles) = unsigned(spi_length) then
          -- on renvoie un seul mot de 32 bits contennant soit la donne ecrite
          -- en mode 'write', soit la donne lue en mode 'read'
          -- l'emplacement de l'octet depends du mode 24 ou 32 bits (spi_length)
          if unsigned(spi_length) = 24 then
            data_write(7 to 15) <= spi_ld & spi_data_out(16 to 23);
          elsif unsigned(spi_length) = 32 then
            data_write(7 to 15) <= spi_ld & spi_data_out(24 to 31);
          else
            data_write <= (others => '1');
          end if;
```

Figura 17: Código software de gestión de datos sobre uB y bloque VHDL de configuración de periféricos

y uB. Este valor nulo sirve de reset inicial permitiendo posicionar los índices de almacenamiento de los valores siguientes. A partir de este punto, la rutina de interrupción escruta la etiqueta de los valores recibidos sobre 32 bits. Si dicha etiqueta, contenida en los 16 bits de mas peso es igual a un valor predeterminado, los datos recibidos son almacenados de manera consecutiva en el array de configuración, hasta un máximo de 16 posiciones. Una vez finalizada la recepción, el flag "ExecuteAction" se posiciona a un valor alto, lo que habilita la ejecución de la acción predeterminada en la instrucción recibida. Esta instrucción aparece en la posición 0 del array de control, siendo las posiciones sucesivas parámetros para este comando. A partir de aquí, el bucle while puede proceder a decodificar este comando, transfiriendo el control a una subrutina específica.

A modo de ejemplo, se han desarrollado una serie de subrutinas que implementan la misma funcionalidad que en el caso de una implementación únicamente en código VHDL (figura 18). La primera de ellas devuelve hacia el colector de datos el valor de los registros de control recibidos (comando=0x01, rutina "readback-reg-bank"), previa conversión a un formato de 32 bits, además de posicionar el flag "ExecuteAction" a un valor de 0, lo que inhabilita la ejecución de una nueva subrutina antes de recibir una nueva interrupción. Este código permitirá repetir los tests presentados en las secciones anteriores de esta memoria, verificando que la transmisión y reenvío iterativo de informaciones de configuración desde / hacia el colector no provoca errores de comunicación, punto fundamental para cualquier implementación posterior.

La segunda rutina que se ha desarrollado devuelve un contador incremental de forma continuada, lo que permite verificar el correcto funcionamiento de la salida de datos desde uB al poder verificar el valor de esos datos. Esto es así en primer lugar porque al ser datos conocidos, su valor puede ser verificado desde el colector, verificando la ausencia de errores. En segundo lugar, este test permite

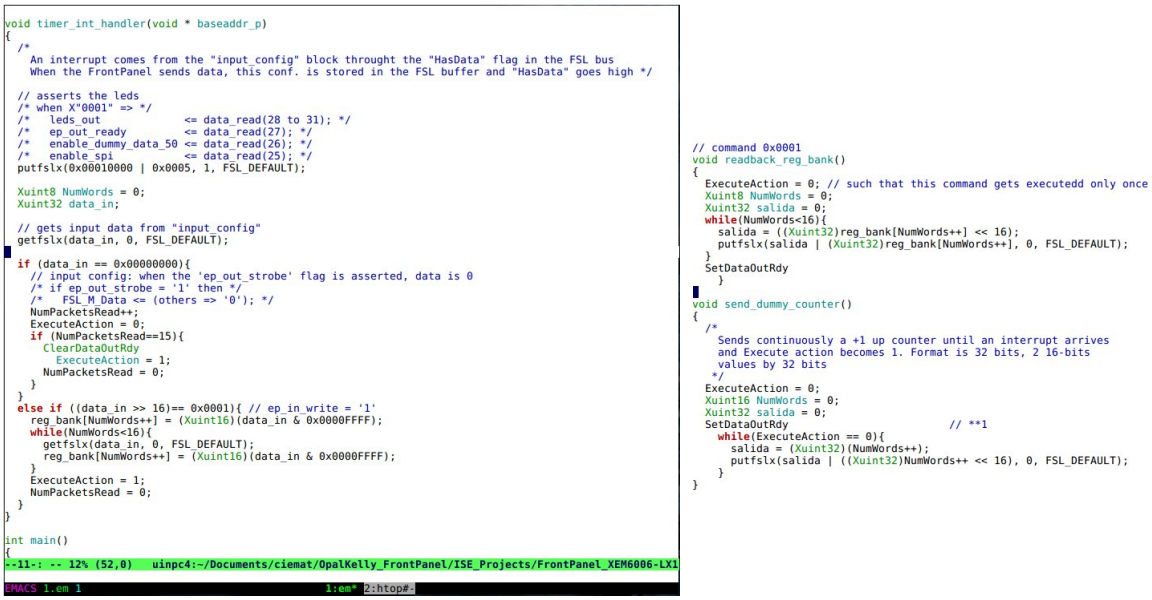


Figura 18: Rutinas de interrupción y reenvío de datos

averiguar cual es la tasa de recuperación de datos, que podrá ser comparada con el caso precedente de implementación únicamente en código VHDL. Se observa como la tasa de transferencia medida en este caso es equivalente al caso anterior, unos 30 MB/s.

Se ha desarrollado un tercer test con un propósito algo mas práctico. En este caso se procede a recuperar datos provenientes de un contador incremental por paquetes de tamaño fijo. El objetivo de este test es el de verificar el funcionamiento del protocolo de transferencia de datos, caracterizándolo de manera a incluirlo en la versión definitiva del sistema. Esta estrategia de recuperación de datos ha sido elegida como compromiso entre el tamaño de los buferes intermedios y la velocidad de recuperación de datos. Mas en detalle, se utilizará un bloque de memoria RAM de 4096 posiciones y 32 bits, y que formará parte del bus FSL de salida de datos. Esto permite enviar paquetes de 16 KB desde uB. La hoja de especificaciones de FP indica que, para tamaños de bloque internos a FP de 1 KB, dicho tamaño permite una salida de datos de hasta 10 MB/s. Esto se ha verificado experimentalmente, implementando una rutina en c de envío de datos por bloques de 16 KB, que al ser recuperados desde el colector permiten medir la tasa esperada. Además, se puede comprobar una tasa de error en la recuperación de datos nula.

Cabe recordar aquí que el propósito de este sistema es el de poder procesar datos en línea y de manera accesible en un sistema embebido, reduciendo la cantidad de informaciones transferidas, por lo que no se desea aprovechar la velocidad máxima de transferencia permitida por el bus usb. En caso de ser necesario aumentar la velocidad de lectura de datos bastaría con incrementar el tamaño del búfer intermedio: un tamaño de búfer de 64 KB permitiría aumentar la tasa hasta alcanzar unos 25 MB/s. En ese caso, los recursos utilizados aumentarían, limitando su disponibilidad para tareas de procesado. Para esta aplicación se ha deseado reservar estos recurso, y el tamaño de bloque se ha limitado a 16 KB.

Siguiendo con la misma lógica, y en el caso en el que se deseen gestionar bloques de datos de tamaño superior, se dispone de una memoria DDR2 de 128 MB de capacidad. Esta memoria permitiría el desarrollo de aplicaciones avanzadas (construcción de histogramas a partir de los datos procesados, etc.) que requieran un uso intensivo de memoria. Con el objetivo de investigar

y caracterizar el funcionamiento del conjunto se ha desarrollado un último test, consistente en la generación de datos conocidos y que serán escritos en memoria DDR2. Posteriormente, dichos datos son transferidos hacia el colector. De nuevo, se observa una tasa de errores nula. Sin embargo, en este caso la tasa de transferencia de datos disminuye en un factor 2, alcanzándose tan solo una velocidad de transferencia de alrededor de 5 MB/s. Esta ralentización sea probablemente debida a la velocidad de acceso a la memoria DDR2, que se ha implementado como un bus bidireccional de 32 bits.

En conclusión, los tests anteriores han permitido verificar la validez de la plataforma desarrollado, permitiendo una mejor comprensión y caracterización del sistema. El código desarrollado puede ser la base a partir de la cual se desarrollen aplicaciones mas complejas. Dichos tests permiten poseer un código de gestión de cada uno de los periféricos vistos hasta ahora. Además, han mostrado cuales pueden ser las posibles limitaciones del sistema y en que sentido se debe de avanzar de cara a mejorarlo.

4. FMC112

Llegados a este punto, en el cual ya se posee las herramientas de gestión y desarrollo tanto de la placa base del sistema como del colector y gestor de datos, se puede proceder a implementar la parte del desarrollo que tienen en cuenta la tarjeta mezzanine de muestreo de datos (FMC112, figura 19). Esta parte implica tanto la comunicación (lectura y escritura de registros de configuración), como la recuperación de muestras. En ambos casos se partirá de la arquitectura desarrollada hasta este momento. De esta manera, será el procesador embebido uB quién comunique con la placa mezzanine, intercambiando informaciones. Al ya existir una comunicación entre el colector de datos y uB, esta sección se centrará en la segunda mitad del desarrollo, que tendrá por finalidad el hacer que el usuario final del sistema tenga la impresión de comunicar directamente con el hardware de muestreo desde el software correspondiente en el colector de datos, o bien poder gestionar el mismo dicha comunicación a partir de una biblioteca de rutinas en c desarrolladas para uB. Con este objetivo, esta sección comenzará por describir los recursos existentes sobre la tarjeta muestreadora, procediendo a continuación a describir el desarrollo realizado de cara a su instrumentación e integración en el sistema de toma de datos.

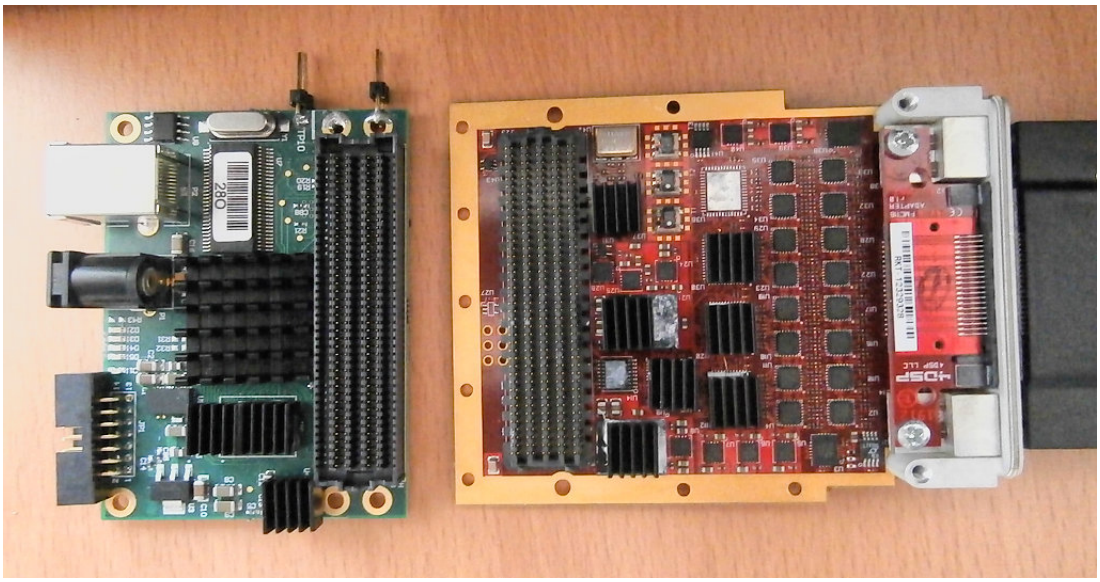


Figura 19: Tarjetas portadora Shutle LX1 (izquierda) y mezzanine FMC 112 (derecha), ambas compatibles con el estándar FMC LPC. Se observa la presencia de dos pines adicionales que permitirán alimentar la tensión de 12 V. a la tarjeta mezzanine a partir de la placa base.

4.1. Descripción del hardware

La tarjeta FMC112 (distribuida por la sociedad 4DSP) posee las características típicas necesarias a un frontal analógico para aplicaciones de instrumentación en física nuclear, pudiendo ser utilizado igualmente en otros ámbitos (figura 20). Como principal característica destaca la inclusión de un conector FMC (ANSI/Vita 57.1 [8]) LPC, lo que permite que esta tarjeta sea acoplada como tarjeta mezzanine o otro módulo portador. Incluye 12 canales de muestreo de señales analógicas a 125 MHz, con 14 bits de precisión, lo que la hace adecuada para un gran rango de aplicaciones (3 X LTC2175 ADC). En la figura 19 se observa la ausencia de uno de los ADC que permitirían operar esta tarjeta con 16 canales. Las señales de entrada deben poseer un rango de 1 o 2 Vpp y un acoplo

DC, pudiendo ser compensado el offset de entrada (LTC2656). La terminación de entrada es de 50 Ohms.

Esta tarjeta está completamente alimentada a través del conector FMC, siendo compatible con un nivel de Vadj de 2.5 V. El estándar FMC impone un nivel de tensión de 12 V. desde la portadora hacia la mezzanine. Esto se ha conseguido utilizando una fuente conmutada (Astec DPT-52M) de potencia capaz de alimentar ambas tarjetas, además del módulo colector de datos, haciendo el sistema mas compacto. Un ejemplo de utilización se detalla en la sección 1 de este documento.

La tarjeta FMC112 posee entradas y salidas tanto de señal de trigger como de señal de reloj de muestro o sincronización, lo que la hace muy flexible. Las señales de trigger son directamente accesibles desde la portadora, mientras que las señales de reloj son manipuladas desde un módulo específico de gestión y distribución de árboles de reloj, ofreciendo varias posibilidades de diseño (AD9517).

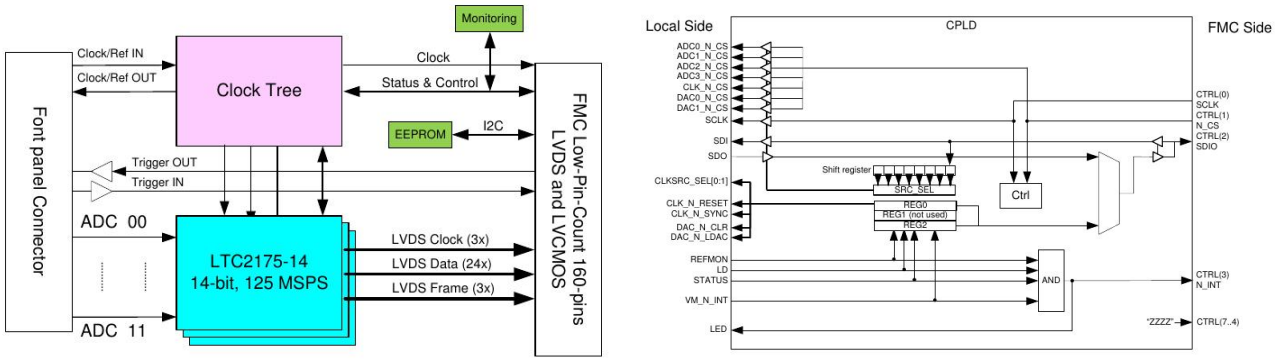


Figura 20: Esquema de bloques de la tarjeta FMC 112 (izquierda) y diagrama de la comunicación con la tarjeta portadora por medio de spi

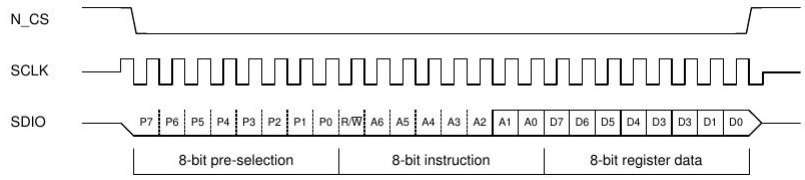


Figura 21: Cronograma spi de comunicación con la tarjeta FMC 112

4.2. Control de la tarjeta

Como se vió en la sección anterior, existen tres componentes fundamentales sobre la tarjeta muestreadora: LTC2175 ADC, el gestor de reloj AD9517 y el DAC LTC2656. Todos ellos poseen en común el poder ser accesibles a través de un puerto spi serie (figura 20). Con el objetivo de minimizar el número de conexiones necesarias para controlar la tarjeta, se ha incluido un CPLD simple cuya única finalidad es la de transmitir órdenes desde / hacia estos tres componentes a partir de un maestro, que podrá dirigirse a todos ellos utilizando un mínimo de señales. Dentro del propio protocolo serie se incluye la dirección del componente de destino, así como el sentido en el que se ejecuta la comunicación (instrucción de lectura o escritura). De esta manera, el CPLD aparece únicamente como distribuidor de órdenes. La figura 21 ilustra este principio, en el que se puede transmitir una instrucción sobre 8 bits, conteniendo igualmente un dato de 8 bits tanto en lectura como en escritura. El destino aparece en los 8 bits de mas peso (bytes de preselección), dando un total de 24 bits sobre la señal SDIO. El byte de preselección se define de la manera siguiente

- CPLD 0x00
- LTC2175 1 0x80 (A/D channels 00 to 03)
- LTC2175 2 0x81 (A/D channels 04 to 07)
- LTC2175 3 0x82 (A/D channels 08 to 11)
- LTC2175 4 0x83 (A/D channels 12 to 15)
- AD9517 0x84
- LTC2656 1 0x85
- LTC2656 2 0x86

El propio CPLD posee una serie de registros internos (figura 22) que permiten tanto configurar el modo de operación como monitorizar el estado de funcionamiento de la tarjeta mezzanine (fuente de reloj, status, etc.).

Bit nr.	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Name	'0'	SYNC	CLKR	LDAC	DACR	Reserved	CLKSRC	

Bit nr.	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Name	Reserved			IRQ	VM	STATUS	LD	REFMON

Bit nr.	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Name	Reserved			LED_SEL				

Figura 22: Mapeado de los registros de control de la tarjeta FMC112

Como se vió en la sección anterior, existen tres componentes fundamentales sobre la tarjeta muestreadora: LTC2175 ADC, el gestor de reloj AD9517 y el DAC LTC2656. Todos ellos poseen en común el poder ser accesibles a través de un puerto spi serie (figura 20). Con el objetivo de minimizar el número de conexiones necesarias para controlar la tarjeta, se ha incluido un CPLD simple cuya única finalidad es la de transmitir órdenes desde / hacia estos tres componentes a partir de un maestro, que podrá dirigirse a todos ellos utilizando un mínimo de señales. Dentro del propio protocolo serie se incluye la dirección del componente de destino, así como el sentido en el que se ejecuta la comunicación (instrucción de lectura o escritura). De esta manera, el CPLD aparece únicamente como distribuidor de órdenes. La figura 21 ilustra este principio, en el que se puede transmitir una instrucción sobre 8 bits, conteniendo igualmente un dato de 8 bits tanto en lectura como en escritura. El destino aparece en los 8 bits de mas peso (bytes de preselección), dando un total de 24 bits sobre la señal SDIO. El byte de preselección se define de la manera siguiente

A partir de las especificaciones anteriores, y basándose en el código presentado en las secciones anteriores de esta memoria, ha sido posible acceder a los periféricos presentes en la tarjeta mezzanine. Par ello se ha desarrollado un código tanto software (lenguaje c) como hardware (vhdl) que realiza las funciones de interfaz entre el procesador microBlaze y los recursos existentes en la placa FMC112. Este código se ilustra en la figura 23.

Con el objetivo de simplificar la lógica utilizada dentro de la FPGA, la parte hardware se ha implementado en el interior del bloque de salida de datos desde uB a través del puerto FSL utilizado, añadiendo nuevas funcionalidades a las ya existentes sin bloquear un nuevo puerto FSL.

Se ha optado por esta alternativa de diseño en lugar de recurrir a recursos existentes en la plataforma de desarrollo que acompaña a uB (EDK). La razón es un mayor dominio sobre los recursos y la funcionalidad implementadas, así como una reducción del tamaño del diseño, aún a cambio quizás de una mayor complejidad, a la que no será expuesto el usuario final del sistema, para quién los detalles deberían de resultar transparentes. De esta manera el procesador puede transmitir directivas codificadas conteniendo tanto la el byte de preselección, como la dirección del registro y el dato que se desee tanto escribir como leer en cualquiera de los periféricos. Estas directivas se transmiten desde uB depositando una serie de bytes en la memoria de intercambio con el periférico, que procede a decodificar estas informaciones, interpretándolas y ejecutando la orden correspondiente, en este caso, el envío de datos sobre el bus spi, generando las señales de reloj y de habilitación de la transmisión características de este protocolo. Los datos se transmiten (o se adquieren, dependiendo de si se trata de una operación de lectura o bien de escritura) a baja velocidad (de unos cuanto MHz), ya que la configuración de la tarjeta mezzanine no es una tarea prioritaria que deba de ejecutarse a una velocidad superior. Se ha tenido en cuenta el hecho de que en esta implementación, la línea SDIO es bidireccional, por lo que se debe de gestionar el estado de alta impedancia en salida de esta línea. Por simplicidad, esta gestión se realiza desde el nivel superior del diseño con una señal *spi-dio-t* que será sintetizada como un búfer de tres estados sobre la señal *spi-dio-o*.

```
#include "Interrupts.h"
#include "ADC_Setup.h"

/*  Configuration des ADC

    - LTC2175 #1    0x80    (A/D channels 00 to 03)
    - LTC2175 #2    0x81    (A/D channels 04 to 07)
    - LTC2175 #3    0x82    (A/D channels 08 to 11)
    - LTC2175 #4    0x83    (A/D channels 12 to 15)

*/
Xuint8 Config_ADC(Xuint8 Read_Write, Xuint8 Instruction, Xuint8 Data, Xuint8 ADC_nb){
  Xuint32   LastInterruptData_saved = 0;
  // i) send command
  SendSpiData(0x8000 | (Xuint16)Instruction | (Xuint16)ADC_nb << 8 | (Xuint16)Read_Write << 7,// MSB
          (Xuint16)Data << 8,                                                              // LSB
          0x18);
  // ii) readback
  if (Read_Write == cpld_write){ // if write
    LastInterruptData_saved = LastInterruptData;
    // force read
    SendSpiData(0x8000 | (Xuint16)Instruction | (Xuint16)ADC_nb << 8 | (Xuint16)cpld_read << 7,
        0xFFFF, // peu importe
        0x18);
    // check write
    if (LastInterruptData != LastInterruptData_saved)
      return -1;
  }
  return 0;
}

Xuint8 SoftReset_ADCs(){
  Xuint8 i;
  // REGISTER A0: RESET REGISTER (ADDRESS 00h)
  // 0 = Not Used
  // 1 = Software Reset. All Mode Control Registers are Reset to 00h. The ADC is Momentarily Placed in SLEEP Mode.
  // This Bit is Automatically Set Back to Zero at the End of the SPI Write Command.
  // The Reset Register is Write Only.
  // This Bit is Automatically Set Back to Zero at the End of the SPI Write Command.
  for (i=0;i<NbADCs;i++)
      // cette fonction doit retouner -1 car registre 0 es en lecture seule
      Config_ADC(0, 0x00, w0_reset_on, i);
  return 0;
}

Xuint8 Disable_ADCs(){
  Xuint8 i;
  // disable ADCs Sleep Mode. All Channels are Disabled
  for (i=0;i<NbADCs;i++){
    if (Config_ADC(0, 0x01, w1_sleep  , i) != 0);
      return -1;
  }
  return 0;
}

Xuint8 Enable_ADCs(){
  Xuint8 i;
  for (i=0;i<NbADCs;i++){
    if (Config_ADC(0, 0x01, 0x00, i) != 0)  // disable ADCs
      return -1;
  }
  return 0;
}

Xuint8 DisableOutputs_ADCs(){
  Xuint8 i;
  // OUTPUT Output Disable Bit
```

```
process (FSL_S_Clk) is
begin
  if FSL_S_Clk'event and FSL_S_Clk = '1' then
    -- arret et reset au repos
    if enable_spi = '0' then
      spi_counter  <= (others => '1');
      spi_dio_o    <= '0';
      spi_dio_t    <= '1';
      spi_disable  <= '0';
      data_write   <= (others => '0');
      enable_write <= '0';
      spi_data_out <= (others => '0');
    else
      if spi_disable = '0' then
        spi_counter <= spi_counter + 1;
        -- condition d'arret
        if unsigned(spi_cycles) = unsigned(spi_length) then
          -- on renvoie un seul mot de 32 bits contennant soit la donne ecrite
          -- en mode 'write', soit la donne lue en mode 'read'
          -- l'emplacement de l'octet depends du mode 24 ou 32 bits (spi_length)
          if unsigned(spi_length) = 24 then
            data_write(7 to 15) <= spi_ld & spi_data_out(16 to 23);
          elsif unsigned(spi_length) = 32 then
            data_write(7 to 15) <= spi_ld & spi_data_out(24 to 31);
          else
            data_write <= (others => '1');
          end if;
          enable_write <= '1';
          spi_disable  <= '1';
        end if;
        -- en mode 'read' on passe en haute impedance
        if spi_read_write_inst = '1' and unsigned(spi_cycles) >= to_unsigned(16, 6) then
          spi_dio_o <= '0';
          spi_dio_t <= '1';
        else
          spi_dio_t <= '0';
          spi_dio_o <= spi_data_in(to_integer(unsigned(spi_cycles)));
        end if;
        --
        if spi_counter(4 downto 0) = "10010" then
          spi_data_out(to_integer(unsigned(spi_cycles))) <= spi_dio_i;
        end if;
      else
        spi_counter  <= (others => '1');
        spi_dio_o    <= '0';
        spi_dio_t    <= '1';
        data_write   <= (others => '0');
        enable_write <= '0';
      end if;
    end if;
  end if;
end process;
spi_clk <= spi_counter(4);
spi_cs  <= spi_counter(11);
```

Figura 23: Código de gestión de la tarjeta mezzanine. A la izquierda el código software en lenguaje c ejecutado desde microbaze; a la derecha el código vhdl de acceso al puerto spi

La segunda parte del desarrollo realizado se ilustra en la parte izquierda de la figura 23. Esta parte implica únicamente desarrollo de código en lenguaje c a partir de las rutinas de bajo nivel de comunicación dadas para intercambiar datos con un puerto FSL. Estas rutinas permiten acceder y depositar palabras de 32 bits a través de una memoria FIFO intermedia, de dimensión reducida de nuevo con el objetivo de minimizar los recursos necesarios. A un nivel superior, se han implementado una serie de bibliotecas correspondientes a los componentes sobre la tarjeta mezzanine (ADC, PLL, etc.) que facilitan el acceso a estos periféricos desde el procesador. Así por ejemplo aparecen rutinas que tienen un significado preciso relativo a cada componente: habilitación o deshabilitación, reset,

configuración de características como la frecuencia de funcionamiento, el nivel de offset en salida de los DACS, etc. El objetivo de esta manera de proceder es proveer a un eventual usuario del sistema unas bibliotecas de configuración de periféricos que faciliten el desarrollo de aplicaciones específicas de forma sencilla.

De manera similar a la manera en que se procedió anteriormente, se han desarrollado también una serie de rutinas sistemáticas de test y verificación del buen funcionamiento de esta comunicación entre el procesador uB y la tarjeta muestreadora. El objetivo aquí ha sido el mismo que en el caso anterior, asegurarse de que se posee un protocolo robusto y fiable de transmisión y recepción de informaciones de configuración, desarrollando la noción de fiabilidad en el intercambio de datos entre ambos. De esta manera, se pretende asegurarse de que no se está programando un componente específico de manera incorrecta, lo que podría tener consecuencias peligrosas para el funcionamiento de la tarjeta (origen de las fuentes de reloj contradictorias, frecuencias de muestreo, número de componentes activos, consumo de la tarjeta, etc.). Estas rutinas de verificación se han ejecutado de manera sistemática sobre un número elevado de ciclos, comparando los datos transmitidos y los recibidos (valores de los registros en los componentes). Así mismo, se han usado los valores de los registros del propio CPLD como criterio de fiabilidad para esta comunicación.

Por último, y desde el momento en el que se es capaz de comunicar con los componentes presentes sobre la placa muestreadora de manera fiable, también será automáticamente posible el combinar esta capacidad con todo lo presentado hasta este momento, la posibilidad de intercambiar informaciones entre el colector de datos y el procesador embebido. Finalmente, llegados a este punto, se pueden combinar las dos funcionalidades anteriores de manera que sea también posible el configurar la placa muestreadora FMC112 desde el software de control de la adquisición. Para ello será suficientes con transmitir los valores adecuados hacia uB, que los reenviará a su vez hacia el periférico adecuado. De la misma manera es también posible el solicitar el valor de cualquiera de los registros en los componentes de la tarjeta, verificando que las informaciones se han transmitido de manera correcta.

4.3. Procesado de datos

Hasta este momento, el objetivo del desarrollo presentado ha sido el de poder implementar una lógica flexible, modular y lo más sencilla posible que permita la implementación de un sistema de toma de datos de manera robusta. Desde un principio, se ha sido muy cuidadoso con el uso de los recursos necesarios, que han sido reducidos al máximo con el objetivo de liberar la mayor cantidad de lógica posible de cara a poder ser utilizada en tareas de procesamiento de datos en línea. Llegados a este punto, el sistema es operacional y permite el intercambio de informaciones entre el colector de datos, Cubox, y el procesador embebido en la placa base Shuttle LX1. Se pueden además configurar de manera adecuada los modos de funcionamiento de la tarjeta mezzanine muestreadora FMC112 (numero de canales, etc.). En esta sección se describirán las tareas de procesado que es posible llevar a cabo como corolario de todo el desarrollo anteriormente descrito.

Las muestras de cada canal electrónico se entregan en salida de los componentes ADC sobre dos pares de líneas diferenciales, como se muestra en la figura 24, aunque este formato e salida de datos es configurable. Con el objetivo de almacenar estas muestras y poder proceder a su tratamiento, se hace necesaria una arquitectura modular y de fácil acceso, que respete los principios introducidos hasta aquí y sea compatible con ellos.

Se ha optado por una opción de diseño basada en el uso de nuevo del puerto FSL, abundante en el procesador uB, y que maximiza la tasa de transferencia de informaciones en aplicaciones de flujo de datos. Varias alternativas son evidentemente posible. Varias de estas alternativas han sido desarrolladas es este trabajo, y cada una de ellas posee sus ventajas e inconvenientes. La mayoría de

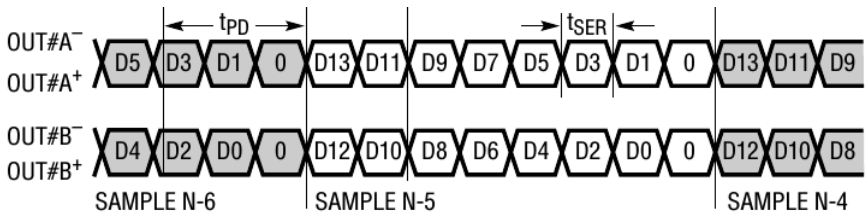


Figura 24: Cronograma de las muestras desde el ADC.

las que se han seguido en este proyecto se fundamentan en el almacenamiento de muestras previa transferencia al procesador. Así por ejemplo, se ha desarrollado el almacenamiento de muestras en un búfer circular implementado a partir de memorias BlockRAM. Estas memorias son cíclicas y contienen las muestras recibidas en los últimos 10 us para tamaños de memoria de 1024 posiciones. El procesador puede activar o desactivar este almacenamiento de datos basándose en informaciones externas (triggers, por ejemplo). Además, podrá solicitar la recuperación de estas muestras desde la memoria BlockRAM para su procesado de manera optimizada, al ser ejecutada cada instrucción *get* sobre tamaños de palabra de 4 bytes. Este modo de funcionamiento puede ser utilizado por ejemplo para una aplicación sencilla en la cual se requiera implementar un osciloscopio digital de precisión. En este caso las muestras son transferidas sin modificación alguna hacia el colector de datos. Esta transferencia puede sincronizarse haciendo uso del trigger externo existente sobre la tarjeta mezzanine, y que podría provenir de un equipo exterior, o bien podría ser generado dentro del sistema. Además, y con el objetivo de optimizar el rendimiento y minimizar el tiempo muerto entre triggers sucesivos, las muestras de varios canales pueden ser transferidas temporalmente hacia la memoria DDR2 de 128 MB. Al llenarse esta, su contenido puede ser volcado hacia el colector. Esta estrategia permite gestionar situaciones en las que se requiera almacenar datos durante periodos de tiempo cortos (del orden del segundo), seguidos por periodos prolongados de interrupción de la señal, como es el caso en instalaciones donde la toma de datos se sincroniza con periodos en los que generación de haz de partículas no es continuado. Hay que recordar que esta estrategia puede ser desarrollada por un usuario del sistema desarrollando únicamente código para uB, puesto que todas las rutinas de acceso a los recursos han sido previstas para ello.

Como variación del método anterior, se puede proceder a implementar una toma de datos síncrona sobre todos los canales en paralelo, o bien gestionando cada uno de ellos por separado. Para ello bastaría con redimensionar correctamente el tamaño de las memorias utilizadas. Para la gestión de los eventos de trigger se procede de la misma manera, y cada canal puede producir un trigger separado o puede preferirse tomar cualquiera de ellos como indicativo de un evento interesante (caso de detectores multicanal, donde se desea almacenar el estado de todos los canales simultáneamente). El trigger es visto por uB como una interrupción que posee una firma particular, que será decodificada: un esquema one-hot sobre 16 bits permite reconocer el origen del trigger y obrar en consecuencia. Hay que recordar aquí que se ha deseado no utilizar un controlador de interrupciones, diseñando uno propio con el objetivo de nuevo de tener un control absoluto sobre la funcionalidad y los recursos utilizados.

Una segunda alternativa de diseño mas ambiciosa que la anterior consiste en no recuperar las muestras de los adc, sino proceder a su procesado en línea de la manera presentada en la referencia [5]. En este caso se han desarrollado una serie de bloques funcionales (figura 25) en lenguaje VHDL consistentes en la implementación de filtros de deconvolución de tipo recursivo IIR [6]. Estos filtros procesan en el dominio temporal las informaciones contenidas en las señales de entrada, procediendo a extraer las informaciones relevantes. Estas informaciones (típicamente la energía contenida en cada

señal electrónica muestreada) son depositadas en las memorias FIFO de cada bus FSL, pudiendo uB recuperar el resultado del procesado para su gestión. Así por ejemplo, uB es capaz de construir, apoyándose en la memoria DDR2, el histograma (espectro) de energías de cada canal, que será recuperado por el colector bajo demanda, pudiendo resetearlo, inicializarlo, calibrarlo, etc. Hay que notar que esta alternativa reduce el ancho de banda necesario en salida del sistema, pudiendo prolongar la toma de datos durante periodos de tiempo prolongados. Además, se puede proceder de manera semejante a la hora de implementar bloques de lógica que generen señales de trigger que provoquen una interrupción del procesador, por ejemplo, aplicando señales de umbral configurables sobre la señal original procesada.

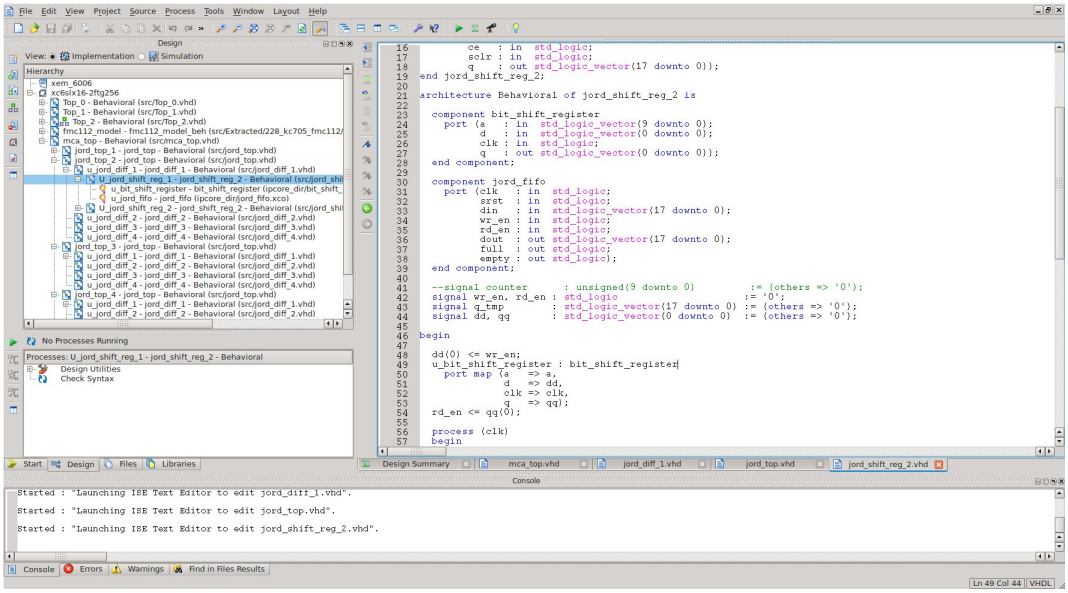


Figura 25: Desarrollo de bloques de procesado por filtrado IIR.

Las alternativas anteriores exigen un desarrollo específico de bloques hardware. Una vez que estos bloques has sido implementados, el usuario final puede elegir una implementación u otra en función de sus necesidades, desarrollando el código apropiado. Sin embargo, esto implica una limitación desde el momento en que el sistema no es capaz de procesar un tipo de señales particulares no previstas inicialmente. Con el objetivo de paliar esta limitación, existe una última alternativa, implementada igualmente en el sistema actual, consistente en autorizar al usuario la concepción de bloques de lógica a medida mediante el uso de herramientas de síntesis a partir de lenguajes de alto nivel, HLS. Típicamente la herramienta Vivado de Xilinx autoriza esta manera de proceder, de forma que el usuario del sistema deberá únicamente respetar una lógica de interfaz con el procesador, siendo esta muy sencilla de tipo FIFO. Desde el procesador podrían bien entregarse las muestras a procesar a este bloque, o bien las muestras pueden ser recogidas desde un bloque de memoria circular como el descrito anteriormente. En cualquier caso, las posibilidades son numerosas y deberán siempre ser adaptadas en función de un compromiso entre el rendimiento y los recursos disponibles.

Por último, es necesario destacar el hecho de que estas metodologías pueden ser extendida en función del número de recursos lógicos existentes. Al ser estos reducidos, todas estas alternativas tendrán evidentemente una limitación que está siendo gestionada en desarrollos posteriores de este sistema, como se menciona en la sección *"2. Conclusión y perspectivas"* de esta memoria.

Parte III
Resultados

1. Montaje Final

Como conclusión del desarrollo mostrado en esta memoria, y a modo de ejemplo práctico de aplicación, se muestra en esta sección como ha llevado a cabo la instrumentación completa de una serie de detectores centelleantes, de uso habitual en el ámbito de la física nuclear. Las imágenes correspondientes a este ejemplo práctico se muestran en el apéndice A de este documento. En este caso el sistema permite instrumentar un tipo de detectores plásticos que son sensibles a un cierto tipo de radiación incidente, y que poseen una respuesta rápida impulsional del orden de varias decenas de nanosegundos. Este tipo de señal poseerá un ancho de banda comprendido en el rango de 40 MHz para el que ha sido diseñado este sistema. Son estas señales las que se muestrean y procesan con ayuda del sistema de toma de datos. En el caso de desear utilizar este sistema para señales mas rápidas, todavía sería posible añadir una etapa preamplificadora.

Este sistema ha sido concebido con una seria de características particulares, de manera que posea las cualidades necesarias requeridas en el ejemplo que aquí se muestra. De esta manera se puede apreciar (figura 26) como en un contexto de instrumentación modular compleja y voluminosa, el poder disponer de un equipo compacto y sencillo aporta un valor añadido importante[8]. El hecho de poder desplazar fácilmente el aparato de medida también abre la puerta a posibilidades de difícil alcance de otra manera: hay que recordar que la instrumentación in-situ muchas veces implica límites prácticos en cuanto al espacio disponible. Así por ejemplo, en este caso se observa como el colector de datos requiere únicamente un acceso remoto por medio de una conexión inalámbrica, disponible por medio de un adaptador wifi económico y poco voluminoso (led azul en las imágenes), soportando conexiones teóricas de hasta 300 Mbps. Dicho acceso inalámbrico autoriza también la recuperación de datos de la adquisición hasta cierto punto. En este caso en particular, este acceso inalámbrico permite una operación remota del sistema, lo que evita no sólo la necesidad de instalar un puesto de operador junto al dispositivo, con la consiguiente aumento en el material requerido (ratón, teclado y sobre todo monitor), sino sobre todo reduce el espacio ocupado requerido para su gestión. Este punto, en el caso de deber operar un sistema sin posibilidad de acceder al sitio experimental puede ser crucial en muchos casos (presencia de radiación de niveles no permitidos, instalación en infraestructuras remotas, etc.). Sin embargo, esto no es una limitación y el sistema evidentemente compatible con cualquier pc de escritorio de gama baja equipado de un simple puerto usb, pudiendo también ser accesible a partir de una conexión fija de 1 Gbps en lugares donde no se disponga de un punto de acceso inalámbrico.

El programa de acceso, gestión y recuperación de datos funciona en modo linea de comandos en forma de script desde el interior de un entorno ipython. Los parámetros de la adquisición son leídos a partir de un simple fichero de texto, de forma que se simplifique su modificación por el usuario. En paralelo, se ha procedido a desarrollar una interfaz gráfica de acceso a la aplicación utilizando la biblioteca gráfica PyQt, de utilización mas sencilla. La aplicación funciona a partir de una distribución Arch Linux específica para sistemas ARM instalada sobre una tarjeta micro SD de 32 GB. La librería FrontPanel para sistemas ARM existe en versión beta desde primeros de año bajo demanda, y no ha dado ningún problema hasta la fecha.

El usuario tiene acceso al sistema a través de una conexión ssh, y el almacenamiento de datos se realiza sobre un disco duro externo conectado al puerto eSata del colector (no mostrado en las

[8] `http://goo.gl/01vp7n`

imágenes), aunque otras configuraciones son posibles. Así por ejemplo es posible el almacenamiento de datos en local hasta los 32 GB disponibles sobre la memoria micro SD, o incluso extender esta capacidad por medio de un disco USB externo. Se ha previsto también el caso en que dicho almacenamiento no fuera suficiente, por ejemplo en el caso de campañas de toma de datos remotas durante periodos de tiempo prolongados (meses), durante los cuales se quiere poder analizar (parte) de los datos colectados de cara a su monitorización con poca latencia temporal. Para gestionar dicha eventualidad, y en el caso en el que el almacenamiento en local estuviese limitado, se ha desarrollado una rutina de envío de datos por red a un almacenamiento remoto, que permite verificar la validez de los datos. Esta funcionalidad ha sido extendida, automatizando la toma de datos remota mediante el uso de tecnologías muy recientes de transferencia de datos punto a punto[9] y de copia de seguridad incremental en intervalos regulares de tiempo sobre servidores distantes[10], complementándose con el envío automático por mail (*mutt*) de los ficheros de log de la adquisición. Todas estas alternativas han sido probadas y funcionan sin problema sobre la plataforma elegida, Cubox.

Respecto a la alimentación del sistema, en las figuras 26 y 27 se puede apreciar como un único transformador de potencia permite alimentar tanto el colector de datos como la electrónica de muestreo y procesado (tanto la placa base como la mezzanine). Ambos elementos intercambian informaciones a través de un cable usb. Respecto al estándar FMC [8], la única divergencia en este montaje reside en el hecho de que la placa portadora no es capaz de proveer por si misma la potencia requerida sobre 12 V. hacia la tarjeta mezzanine, ya que ella misma no requiere más de 5 V para funcionar. Sin embargo si que está previsto el poder hacerlo a partir de una fuente externa (cable rojo en las imágenes), que produzca la tensión y corrientes requeridas, a través de dos puntos específicos sobre la tarjeta base (dos pares de pines en la parte izquierda de la figura 19), de ahí que se haya construido un cable a medida para este instrumento con tres tipos de conexiones distintas, una por cada elemento del sistema.

A partir de las figuras se observa también como la tarjeta mezzanine permite trabajar con señales de hasta doce detectores independientes por medio de un conector compacto que da acceso a un mazo de cables terminados en un conector SMA (este haz de cables se incluye de serie con la tarjeta mezzanine). En las figuras anteriores uno de estos cables se ha conectado a una señal de un detector (BNC) a través de un adaptador SMA-BNC, pudiéndose también trabajar con el estándar LEMO, siendo ambos muy utilizados en instrumentación nuclear. Además se puede observar sobre la tarjeta mezzanine la presencia de dos pares de conectores de entrada y salida de señales de reloj y de trigger (no utilizadas en las imágenes), que permiten extender este sistema a un número mas elevado de canales.

Cabe decir que esta aplicación se lleva a cabo en un entorno de temperatura controlada de unos 20-25 grados centígrados, por lo que la disipación térmica no ha sido un problema hasta ahora. Después de haber consultado este punto con el servicio técnico de SolidRun, éstos afirman haber llevado a cabo tests de conformidad térmica a temperatura ambiente forzando a Cubox a funcionar a pleno rendimiento mediante una decodificación vídeo HD (3 watios de potencia disipada). En estas condiciones la temperatura interna medida es de unos 55 grados centígrados, lo que se corresponde con las medidas realizadas. Aún así, está previsto el integrar los diversos elementos del sistema de toma de datos en una estructura mecánica, que a la vez que proteja al equipo en entornos más severos, permita también una refrigeración constante por flujo de aire mediante un ventilador.

Por último, destacar una vez mas que obviamente este sistema de toma de datos podría de manera sencilla (por diseño) adaptarse a otros ámbitos de investigación que deban de gestionar de manera precisa y con una dinámica importante señales de un ancho de banda de hasta unos 40 MHz, de manera similar a la presentada aquí.

[9] `http://labs.bittorrent.com/experiments/sync.html`
[10] `http://duplicity.nongnu.org/`

2. Conclusión y perspectivas

Como se comentó al inicio de esta memoria, el objetivo de este desarrollo ha sido múltiple y ambicioso. Por un lado se ha pretendido poder desarrollar un instrumento de toma de datos, capaz de equipar un experimento en condiciones reales experimentales, teniendo en cuenta todas las limitaciones y requerimientos que esto exige, como así ha sido. Por otra parte, se ha buscado desde un principio poder concebir un instrumento lo suficientemente polivalente y flexible, permitiendo así poder responder a las necesidades de instrumentación de otro de tipo de experimentos mas allá de los inicialmente previstos.

Este último punto se ha considerado desde un inicio tanto en relación a la elección de componentes como respecto a la arquitectura de los elementos del desarrollo presentados en esta memoria. De esta manera, además de implementar un demostrador completo funcional, se ha pretendido facilitar el acceso a este instrumento a usuarios arbitrarios sin exigirles a cambio un conocimiento avanzado en desarrollo de componentes embebidos y al mismo tiempo sin requerir una modificación sustancial de la aplicación original, lo cual supone en muchas ocasiones una ventaja no despreciable en este tipo de sistemas.

Este objetivo ha podido realizarse gracias a una aproximación mixta, basada en el desarrollo de código a varios niveles diferentes (software de aplicación y de nivel intermedio, hardware mediante hdl y para procesador embebido) y en la existencia de los componentes materiales necesarios comercialmente, elegidos explícitamente debido a su adecuación con las características buscadas para el sistema de toma de datos. Esta manera de proceder es original hasta cierto punto en el sentido en que diverge de la manera habitual de trabajar en ciertos campos de investigación. Muestra así un ejemplo de desarrollo de proyecto instrumental económico, flexible y caracterizado por un tiempo de desarrollo reducido de forma sencilla, y converge hacia una metodología cada vez mas habitual en otros desarrollos actuales.

A partir de esta experiencia pueden extraerse algunas conclusiones interesantes, que serán tanto positivas como negativas en ciertos aspectos. Por un lado cabe mencionar que la elección de los módulos se ha revelado satisfactoria. La tarjeta digitalizadora responde a las necesidades de un abanico de aplicaciones muy amplio, y su flexibilidad (control de la frecuencia, número de canales, etc.) la hacen adecuada a un uso que va mas allá del presentado en este trabajo. La placa base, aún poseyendo limitaciones evidentes, cumple su cometido de núcleo centralizador para un sistema de gama baja. Por último, la CPU colectora de datos es mas que suficiente para desarrollar aplicaciones en este contexto.

Sin embargo, todos estos elementos podrían suponer únicamente una primera iteración funcional operativa para un sistema más ambicioso que supere los límites materiales de la implementación actual: recursos lógicos programables limitados, componentes antiguos, velocidades de transferencia reducidas, etc. Estas limitaciones son significativas relativamente a la aplicación que se persiga y al rendimiento deseado del sistema de toma de datos. Un factor evidente limitativo aparece en relación a la lógica disponible para bloques de procesamiento de datos. Por otro lado, el hecho de desarrollar aplicaciones para un procesador embebido microBlaze no supone un problema mayor. Esto es así desde el momento en el que cada aplicación independiente puede ser precompilada en formato ELF, almacenada en disco localmente y enviada de forma exclusiva hacia la placa en la etapa de inicialización del hardware, lo cual no supone ninguna limitación funcional. Hay que recordar que el código de configuración de la FPGA Spartan6 no se almacena localmente, sino que se transmite desde la aplicación en el colector de datos. Así por ejemplo, una aplicación que desee recuperar únicamente muestras puede extenderse sobre todo el espacio disponible de ejecución de instrucciones. Si se desea ejecutar una aplicación distinta, la primera se elimina y la segunda puede de nuevo aprovechar todos los recursos: el tiempo de reconfiguración no es una significativo para

esta aplicación. Hay que señalar de la misma manera que este mismo principio podría aplicarse a los bloques de procesamiento de datos, al dimensionado de microBlaze y por extensión al resto del diseño de la FPGA, utilizando un diseño diferente para cada aplicación (tamaños de fichero ELF distinto, FIFO de intercambio de datos -velocidad de recuperación de datos- de tamaño diferente, etc.). Sin embargo, aun siendo posible, esta forma de proceder limitaría el número de usuarios del sistema a aquellos con capacidad para implementar estas modificaciones, lo cual podría suponer una fuente de limitaciones distinta.

Otra posible limitación de la implementación actual proviene del hecho de haber incluido un procesador software en el sistema. Esta elección viene motivada por el deseo de permitir un acceso simplificado al sistema a un número grande de usuarios, aumentando la flexibilidad del sistema. Hay que recordar que la frecuencia de funcionamiento de microBlaze en esta aplicación es de 100 MHz, superior por un factor 2 a la frecuencia del dominio de reloj de la lógica USB. Además, la memoria DDR 2 sobre el módulo base es de 200 MHz. Es evidente que un rendimiento superior podría haberse obtenido con una implementación diferente basada únicamente en un diseño en lenguaje HDL. Sin embargo, esto habría cerrado la puerta a muchos usuarios potenciales del sistema. Por otra parte, la aplicación actualmente en uso es capaz de procesar datos con un margen suficiente, por lo que microBlaze no es un factor limitante, y sus ventajas compensan sus inconvenientes. Por otra parte, queda margen para investigar el origen de los factores limitativos observados hasta este momento, como son la tasa de alrededor de 5 MB/s observada en lectura de datos. Esta ralentización sea probablemente debida a la velocidad de acceso a la memoria DDR2, que se ha implementado como un bus bidireccional de 32 bits. Queda por estudiar el rendimiento para tamaños de bus de 128 bits.

Con el objetivo de sobrepasar cualquier limitación, desde un principio la elección de componentes tuvo en mente etapas de desarrollo posteriores, lo que justifica en parte la elección del formato FMC [8] como estándar de concepción, ya que este estándar autoriza la reutilización de componentes discretos, remplazando únicamente elementos parciales en el sistema. Mas en concreto, varias posibilidades de actualización y mejora del sistema pueden ser conseguidas a partir del montaje actual. La sociedad OpalKelly dispone de módulos con un bus de salida USB3, compatibles con las librerías FrontPanel, incluyendo una actualización prevista para la placa Shuttle LX1 antes de final de año. Esto permitiría una mejora muy importante de rendimiento a cambio de un número de modificaciones mínimas en el diseño actual, tanto software como hardware, lo que abre el abanico de aplicaciones posibles.

Una segunda opción mas ambiciosa, y actualmente bajo desarrollo consiste en reimplementar parte del diseño actual adaptándolo a una arquitectura basada en el uso de placas base de mucho mayor rendimiento. Estas placas base (KC705 de Xilinx, por ejemplo) incluyen conectores FMC LPC y recursos lógicos de una generación mas actual (FPGA Kindex 7), mayores capacidades de memoria mas veloces (DDR 3), además de darse en un formato PCI express. Esta nueva óptica permite una mejora notable en el rendimiento del sistema en lo relativo a los recursos lógicos para procesado de datos, velocidad de ejecución del procesador microBlaze pero sobre todo en lo relativo a la tasa de transferencia de datos, que puede alcanzar fácilmente tasas de 1 GB/s. Estas nuevas características extienden el sistema hacia un número mayor de aplicaciones, basándose siempre en la arquitectura presentada en este proyecto, por lo que se requerirían modificaciones no demasiado importantes para su funcionamiento. Esto permitiría disponer de un procesador software de mayor rendimiento, lo que conllevaría a tiempos de procesado reducidos. Evidentemente, todo lo anterior tendría como contrapartida un aumento del coste del sistema, lo cual puede suponer de nuevo otro tipo de limitación diferente.

Referencias

[1] Luigi Bardelli, Giacomo Poggi, *Digital-sampling systems in high-resolution and wide dynamic-range energy measurements: Finite time window, baseline effects, and experimental tests*. Nuclear Instruments and Methods in Physics Research A 560, 524 2006.

[2] Pil Soo Lee, Chun Sik Lee, Ju Hahn Lee, *Development of FPGA-based digital signal processing system for radiation spectroscopy*. Radiation Measurements Journal 2012.

[3] Hu, Wei; Cho, Y.; Jung, Jin Ho *A FPGA-based high speed multi-channel simultaneous signal acquisition method for PET*. IEEE Nuclear Science Symposium Conference Record (NSS/MIC) 2009.

[4] Cao XiaoQiang ; Zhang Jun ; Yao Weitao ; Ju Mingye *High-Speed and Portable Data Acquisition System Based on FPGA*. Fifth International Conference on Intelligent Networks and Intelligent Systems (ICINIS) 2012.

[5] P. Medina et al. *TNT digital pulse processor*. 14th IEEE-NPSS Real Time Conference 2005.

[6] Valentin T. Jordanov and Glenn F. Knoll, *Digital synthesis of pulse shapes in real time for high resolution radiation spectroscopy*. Nucl. Instr. Meth. A345, 337-345 1994

[7] *Semiconductor Evaluation Leveraging COTS FPGAs and Connectors*. OpalKelly White Paper 2012.

[8] VMEBus International Trade Association (VITA) *ANSI/Vita 57.1 FPGA Mezzanine Card (FMC) Standard, Approved July 2008*. Revisión Febrero 2010.

[9] Opal Kelly's Front Panel User Guide, 2012.

[10] Opal Kelly's Shuttle LX1 User Guide, 2012.

[11] 4DSP FMC112-FMC116 User Manual, 2012.

A. Apéndice A

Este apéndice recoge las imágenes correspondientes al sistema de toma de datos presentado en esta memoria funcionando en laboratorio. En la primera figura se puede observar los tres elementos materiales fundamentales de este desarrollo: i) el colector de datos (cubox), que también hace las veces de entorno de configuración de la electrónica de adquisición; ii) la placa base Shuttle LX1 como plataforma de captura y procesado de las muestras recibidas desde la tarjeta mezzanine iii) FMC112 de 12/16 canales. El equipo instrumenta un detector de partículas (carcasa de aluminio) y permite remplazar una serie de módulos electrónicos utilizados hasta la fecha.

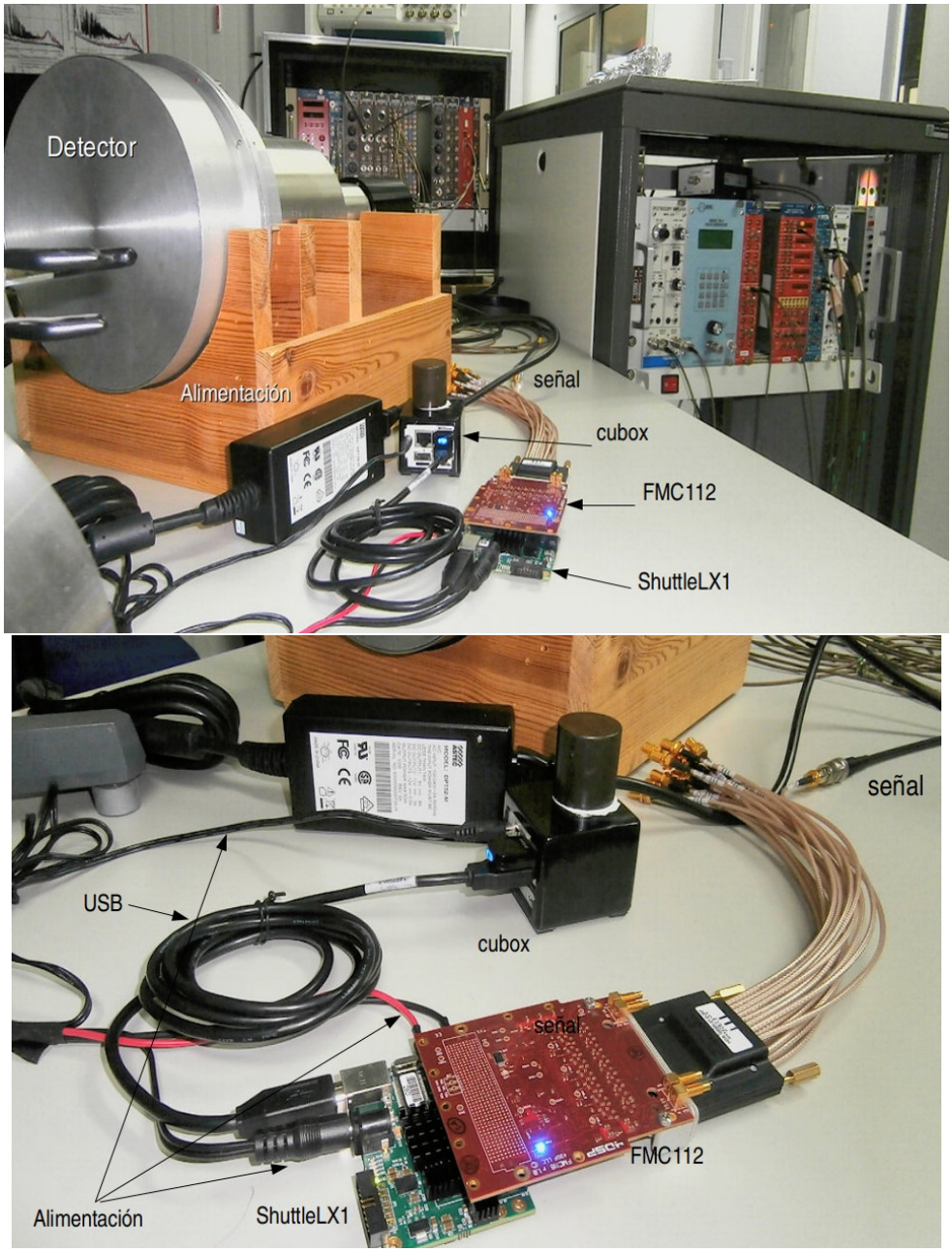


Figura 26: Funcionamiento del sistema de toma de datos en un entorno real en laboratorio.

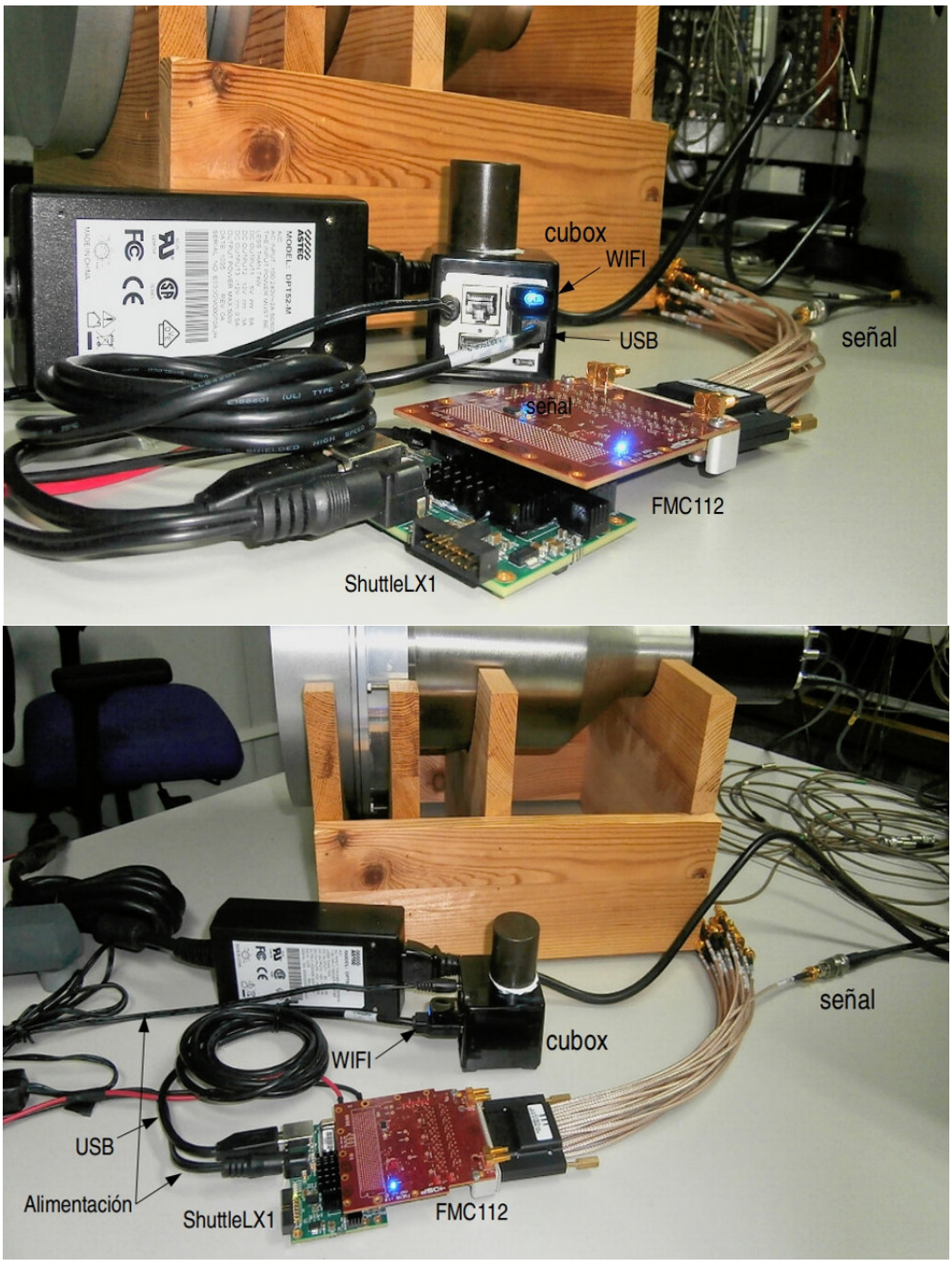


Figura 27: Funcionamiento del sistema de toma de datos en un entorno real en laboratorio (continuación)